

[Unit 1: Introduction]
Artificial Intelligence (CSC 355)

Shiv Raj Pant

Central Department of Computer Science & Information Technology
Tribhuvan University

Intelligence

Scientists have proposed two major “consensus” definitions of intelligence:

(i) from *Mainstream Science on Intelligence* (1994);

A very general mental capability that, among other things, involves the ability to reason, plan, solve problems, think abstractly, comprehend complex ideas, learn quickly and learn from experience. It is not merely book learning, a narrow academic skill, or test-taking smarts. Rather, it reflects a broader and deeper capability for comprehending our surroundings- making sense” of things, or “figuring out” what to do.

(ii) from *Intelligence: Knowns and Unknowns* (1995);

Individuals differ from one another in their ability to understand complex ideas, to adapt effectively to the environment, to learn from experience, to engage in various forms of reasoning, [and] to overcome obstacles by taking thought. Although these individual differences can be substantial, they are never entirely consistent: a given person’s intellectual performance will vary on different occasions, in different domains, as judged by different criteria. Concepts of “intelligence” are attempts to clarify and organize this complex set of phenomena.

Thus, ***intelligence*** is:

- the ability to reason
- the ability to understand
- the ability to create
- the ability to Learn from experience
- the ability to plan and execute complex tasks

What is Artificial Intelligence?

"Giving machines ability to perform tasks normally associated with *human* intelligence."

AI is intelligence of machines and branch of computer science that aims to create it. AI consists of design of intelligent agents, which is a program that perceives its environment and takes action that maximizes its chance of success. With AI it comes issues like deduction, reasoning, problem solving, knowledge representation, planning, learning, natural language processing, perception, etc.

“Artificial Intelligence is the part of computer science concerned with designing intelligence computer systems, that is, systems that exhibit the characteristics we associate with intelligence in human behavior.”

Different definitions of AI are given by different books/writers. These definitions can be divided into two dimensions.

Systems that think like humans	Systems that think rationally
<p>"The exciting new effort to make computers think.....<i>machine with minds</i>, in the full and literal sense." (Haugeland, 1985)</p> <p>"[The automaton of] activities that we associate with human thinking, activities such as decision-making, problem solving, learning....." (Bellman, 1978)</p>	<p>"The study of mental faculties through the use of computational models." (Charniak and McDermott, 1985)</p> <p>"The study of the computations that make it possible to perceive, reason, and act." (Winston, 1992)</p>
Systems that act like humans	Systems that act rationally
<p>"The art of creating machines that perform functions that require intelligence when performed by people." (Kurzweil, 1990)</p> <p>"The study of how to make computer do things at which, at the moment, people are better." (Rich and Knight, 1991)</p>	<p>"Computational Intelligence is the study of the design of intelligent agents." (Poole <i>et al.</i>, 1998)</p> <p>"AI... is concerned with intelligent behavior in artifacts." (Nilsson, 1998)</p>

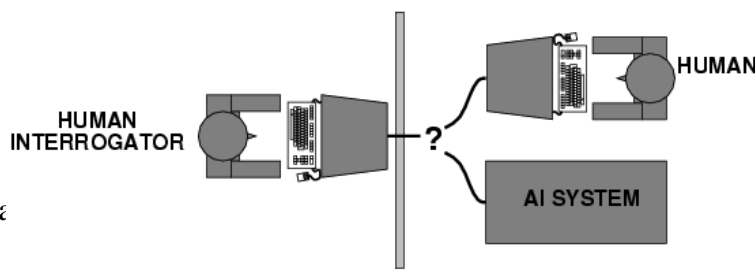
Top dimension is concerned with ***thought processes and reasoning***, where as bottom dimension addresses the ***behavior***.

The definition on the left measures the success in terms of fidelity of *human performance*, whereas definitions on the right measure an *ideal concept of intelligence*, which is called **rationality**.

Human-centered approaches must be an empirical science, involving hypothesis and experimental confirmation. A rationalist approach involves a combination of mathematics and engineering.

Acting Humanly: The Turing Test Approach

The **Turing test**, proposed by Alan Turing (1950) was designed to convince the people that whether a particular machine can think or not. He suggested a test based on indistinguishability from undeniably intelligent entities- human beings. **The test involves an interrogator who interacts with one human and one machine. Within a given time the interrogator has to find out which of the two the human is, and which one the machine.**



The computer passes the test if a human interrogator after posing some written questions, cannot tell whether the written response come from human or not.

To pass a Turing test, a computer must have following capabilities:

- Natural Language Processing: Must be able to communicate successfully in English
- Knowledge representation: To store what it knows and hears.
- Automated reasoning: Answer the Questions based on the stored information.
- Machine learning: Must be able to adapt in new circumstances.

Turing test avoid the physical interaction with human interrogator. Physical simulation of human beings is not necessary for testing the intelligence.

The total Turing test includes video signals and manipulation capability so that the interrogator can test the subject's perceptual abilities and object manipulation ability. To pass the total Turing test computer must have following additional capabilities:

- Computer Vision: To perceive objects
- Robotics: To manipulate objects and move

Thinking Humanly: Cognitive modeling approach

If we are going to say that a given program thinks like a human, we must have some way of determining how humans think. We need to get inside the actual workings of human minds. There are two ways to do this:

- **through introspection:** catch our thoughts while they go by
- **through psychological experiments.**

Once we have precise theory of mind, it is possible to express the theory as a computer program.

The field of cognitive science brings together computer models from AI and experimental techniques from psychology to try to construct precise and testable theories of the workings of the human mind.

Think rationally: The laws of thought approach

Artificial Intelligence

Aristotle was one of the first who attempt to codify the *right thinking* that is irrefutable reasoning process. He gave Syllogisms that always yielded correct conclusion when correct premises are given.

For example:

Ram is a man

All men are mortal

⇒ Ram is mortal

These law of thought were supposed to govern the operation of mind: This study initiated the field of logic. The logicist tradition in AI hopes to create intelligent systems using logic programming. However there are two obstacles to this approach. First, It is not easy to take informal knowledge and state in the formal terms required by logical notation, particularly when knowledge is not 100% certain. Second, solving problem principally is different from doing it in practice. Even problems with certain dozens of fact may exhaust the computational resources of any computer unless it has some guidance as which reasoning step to try first.

Acting Rationally: The rational Agent approach:

Agent is something that acts.

Computer agent is expected to have following attributes:

- Autonomous control
- Perceiving their environment
- Persisting over a prolonged period of time
- Adapting to change
- And capable of taking on another's goal

Rational behavior: doing the right thing.

The right thing: that which is expected to maximize goal achievement, given the available information.

Rational Agent is one that acts so as to achieve the best outcome or, when there is uncertainty, the best expected outcome.

In the “laws of thought” approach to AI, the emphasis was given to correct inferences. Making correct inferences is sometimes part of being a rational agent, because one way to act rationally is to reason logically to the conclusion and act on that conclusion. On the other hand, there are also some ways of acting rationally that cannot be said to involve inference. *For Example, recoiling from a hot stove is a reflex action that usually more successful than a slower action taken after careful deliberation.*

Advantages:

Artificial Intelligence

- It is more general than laws of thought approach, because correct inference is just one of several mechanisms for achieving rationality.
- It is more amenable to scientific development than are approaches based on human behavior or human thought because the standard of rationality is clearly defined and completely general.

Characteristics of A.I. Programs

- **Symbolic Reasoning:** reasoning about objects represented by symbols, and their properties and relationships, not just numerical calculations.
- **Knowledge:** General principles are stored in the program and used for reasoning about novel situations.
- **Search:** a ``weak method'' for finding a solution to a problem when no direct method exists. Problem: *combinatoric explosion* of possibilities.
- **Flexible Control:** Direction of processing can be changed by changing facts in the environment.

Foundations of AI:

Philosophy:

Logic, reasoning, mind as a physical system, foundations of learning, language and rationality.

- Where does knowledge come from?
- How does knowledge lead to action?
- How does mental mind arise from physical brain?
- Can formal rules be used to draw valid conclusions?

Mathematics:

Formal representation and proof algorithms, computation, undecidability, intractability, probability.

- What are the formal rules to draw the valid conclusions?
- What can be computed?
- How do we reason with uncertain information?

Psychology:

Adaptation, phenomena of perception and motor control.

- How humans and animals think and act?

Economics:

Formal theory of rational decisions, game theory, operation research.

Artificial Intelligence

- How should we make decisions so as to maximize payoff?
- How should we do this when others may not go along?
- How should we do this when the payoff may be far in future?

Linguistics:

Knowledge representation, grammar

- How does language relate to thought?

Neuroscience:

Physical substrate for mental activities

- How do brains process information?

Control theory:

Homeostatic systems, stability, optimal agent design

- How can artifacts operate under their own control?

Brief history of AI

- 1943: Warren Mc Culloch and Walter Pitts: a model of artificial boolean neurons to perform computations.
 - First steps toward connectionist computation and learning (Hebbian learning).
 - Marvin Minsky and Dann Edmonds (1951) constructed the first neural network computer
- 1950: Alan Turing's "Computing Machinery and Intelligence"
 - First complete vision of AI.

The birth of AI (1956):

- Dartmouth Workshop bringing together top minds on automata theory, neural nets and the study of intelligence.
 - Allen Newell and Herbert Simon: The logic theorist (first nonnumeric thinking program used for theorem proving)
 - For the next 20 years the field was dominated by these participants.

Great expectations (1952-1969):

- Newell and Simon introduced the General Problem Solver.

Artificial Intelligence

- Imitation of human problem-solving
- Arthur Samuel (1952-) investigated game playing (checkers) with great success.
- John McCarthy(1958-) :
 - Inventor of Lisp (second-oldest high-level language)
 - Logic oriented, Advice Taker (separation between knowledge and reasoning)
- Marvin Minsky (1958 -)
 - Introduction of microworlds that appear to require intelligence to solve: e.g. blocks-world.
 - Anti-logic orientation, society of the mind.

Collapse in AI research (1966 - 1973):

- Progress was slower than expected.
 - Unrealistic predictions.
- Some systems lacked scalability.
 - Combinatorial explosion in search.
- Fundamental limitations on techniques and representations.
 - Minsky and Papert (1969) Perceptrons.

AI revival through knowledge-based systems (1969-1970):

- General-purpose vs. domain specific
 - E.g. the DENDRAL project (Buchanan et al. 1969)

First successful knowledge intensive system.

- Expert systems
 - MYCIN to diagnose blood infections (Feigenbaum et al.)
 - Introduction of uncertainty in reasoning.
- Increase in knowledge representation research.
 - Logic, frames, semantic nets, ...

AI becomes an industry (1980 - present):

- R1 at DEC (McDermott, 1982)
- Fifth generation project in Japan (1981)
- American response ...

Puts an end to the AI winter.

Connectionist revival (1986 - present): (Return of Neural Network):

Artificial Intelligence

- Parallel distributed processing (RumelHart and McClelland, 1986); backprop.

AI becomes a science (1987 - present):

- In speech recognition: hidden markov models
- In neural networks
- In uncertain reasoning and expert systems: Bayesian network formalism

The emergence of intelligent agents (1995 - present):

- The whole agent problem:
“How does an agent act/behave embedded in real environments with continuous sensory inputs”

Applications of AI: (Describe these application areas yourself)

- Autonomous planning and scheduling
- Game playing
- Autonomous Control
- Expert Systems
- Logistics Planning
- Robotics
- Language understanding and problem solving
- Speech Recognition
- Computer Vision

Knowledge:

Knowledge is a theoretical or practical understanding of a subject or a domain. Knowledge is also the sum of what is currently known.

Knowledge is “the sum of what is known: the body of truth, information, and principles acquired by mankind.” Or, "Knowledge is what I know, Information is what we know."

There are many other definitions such as:

- Knowledge is "information combined with experience, context, interpretation, and reflection. It is a high-value form of information that is ready to apply to decisions and actions." (T. Davenport et al., 1998)
- Knowledge is “human expertise stored in a person’s mind, gained through experience, and interaction with the person’s environment.” (Sunasee and Sewery, 2002)
- Knowledge is “information evaluated and organized by the human mind so that it can be used purposefully, e.g., conclusions or explanations.” (Rousa, 2002)

Artificial Intelligence

Knowledge consists of information that has been:

- interpreted,
- categorised,
- applied, experienced and revised.

In general, knowledge is more than just data, it consist of: facts, ideas, beliefs, heuristics, associations, rules, abstractions, relationships, customs.

Research literature classifies knowledge as follows:

Classification-based Knowledge	»	Ability to classify information
Decision-oriented Knowledge	»	Choosing the best option
Descriptive knowledge	»	State of some world (heuristic)
Procedural knowledge	»	How to do something
Reasoning knowledge	»	What conclusion is valid in what situation?
Assimilative knowledge	»	What its impact is?

Knowledge is important in AI for making intelligent machines. Key issues confronting the designer of AI system are:

Knowledge acquisition: Gathering the knowledge from the problem domain to solve the AI problem.

Knowledge representation: Expressing the identified knowledge into some knowledge representation language such as propositional logic, predicate logic etc.

Knowledge manipulation: Large volume of knowledge has no meaning until up to it is processed to deduce the hidden aspects of it. Knowledge is manipulated to draw conclusions from knowledgebase.

Importance of Knowledge:

Learning:

It is concerned with design and development of algorithms that allow computers to evolve behaviors based on empirical data such as from sensor data. A major focus of learning is to automatically learn to recognize complex patterns and make intelligent decision based on data.

Artificial Intelligence

A complete program is said to learn from experience E with respect to some class of tasks T and performance measure P , if its performance at tasks in T , as measured by P , improves with experience E .

[Unit 2: Agents]
Artificial Intelligence (CSC 355)

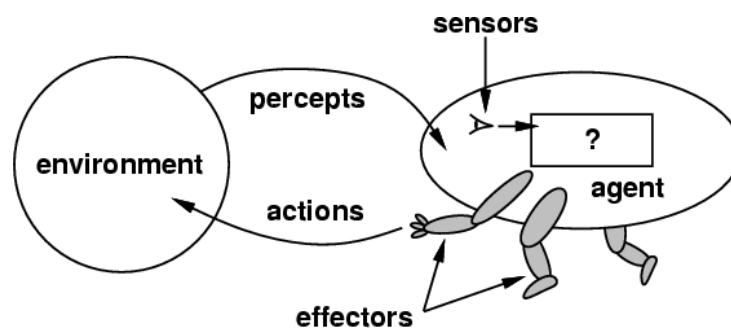
Jagadish Bhatta
Central Department of Computer Science & Information Technology
Tribhuvan University

Intelligent Agents

An Intelligent Agent perceives its environment via sensors and acts rationally upon that environment with its effectors (actuators). Hence, an agent gets percepts one at a time, and maps this percept sequence to actions.

Properties of the agent

- Autonomous
- Interacts with other agents plus the environment
- Reactive to the environment
- Pro-active (goal- directed)



What do you mean, sensors/percepts and effectors/actions?

For Humans

- **Sensors:** Eyes (vision), ears (hearing), skin (touch), tongue (gestation), nose (olfaction), neuromuscular system (proprioception)
- **Percepts:**
 - At the lowest level – electrical signals from these sensors
 - After preprocessing – objects in the visual field (location, textures, colors, ...), auditory streams (pitch, loudness, direction), ...
- **Effectors:** limbs, digits, eyes, tongue,
- **Actions:** lift a finger, turn left, walk, run, carry an object, ...

A more specific example: Automated taxi driving system

- **Percepts:** Video, sonar, speedometer, odometer, engine sensors, keyboard input, microphone, GPS, ...
- **Actions:** Steer, accelerate, brake, horn, speak/display, ...
- **Goals:** Maintain safety, reach destination, maximize profits (fuel, tire wear), obey laws, provide passenger comfort, ...
- **Environment:** Urban streets, freeways, traffic, pedestrians, weather, customers, ...

[Different aspects of driving may require different types of agent programs!]

Challenge!!

Compare Software with an agent

Compare Human with an agent

Percept: The Agents perceptual inputs at any given instant.

Percept Sequence: The complete history of everything the agent has ever perceived.

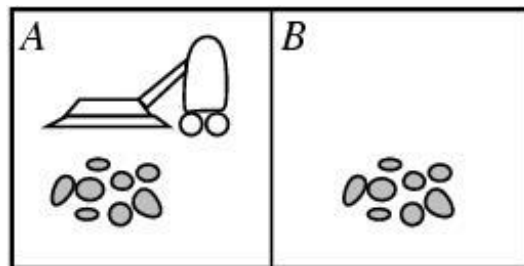
The *agent function* is mathematical concept that maps percept sequence to actions.

$$f : P^* \rightarrow A$$

The *agent function* will internally be represented by the *agent program*.

The agent program is concrete implementation of agent function it runs on the physical *architecture* to produce *f*.

The vacuum-cleaner world: Example of Agent



Environment: square A and B

Percepts: [location and content] *E.g.* [A, Dirty]

Actions: left, right, suck, and no-op

The A one right	Percept sequence	Action	<u>concept</u> of <u>rationality</u> rational agent is that does the thing. – Every entry in the table
	[A,Clean]	Right	
	[A, Dirty]	Suck	
	[B, Clean]	Left	
	[B, Dirty]	Suck	
	

is filled out correctly.

What is the right thing?

- Right action is the one that will cause the agent to be most successful.

Therefore we need some way to measure success of an agent. Performance measures are the criterion for success of an agent behavior.

E.g., performance measure of a vacuum-cleaner agent could be amount of dirt cleaned up, amount of time taken, amount of electricity consumed, amount of noise generated, etc.

It is better to design Performance measure according to what is wanted in the environment instead of how the agents should behave.

It is not easy task to choose the performance measure of an agent. For example if the performance measure for automated vacuum cleaner is “The amount of dirt cleaned within a certain time” Then a rational agent can maximize this performance by cleaning up the dirt , then dumping it all on the floor, then cleaning it up again , and so on. Therefore “How clean the floor is” is better choice for performance measure of vacuum cleaner.

What is rational at a given time depends on four things:

- Performance measure,
- Prior environment knowledge,
- Actions,
- Percept sequence to date (sensors).
-

Definition: *A rational agent chooses whichever action maximizes the expected value of the performance measure given the percept sequence to date and prior environment knowledge.*

Environments

Artificial Intelligence

To design a rational agent we must specify its task environment. Task environment means: PEAS description of the environment:

- Performance
- Environment
- Actuators
- Sensors

Example: Fully automated taxi:

- PEAS description of the environment:

Performance: Safety, destination, profits, legality, comfort

Environment: Streets/freeways, other traffic, pedestrians, weather,, ...

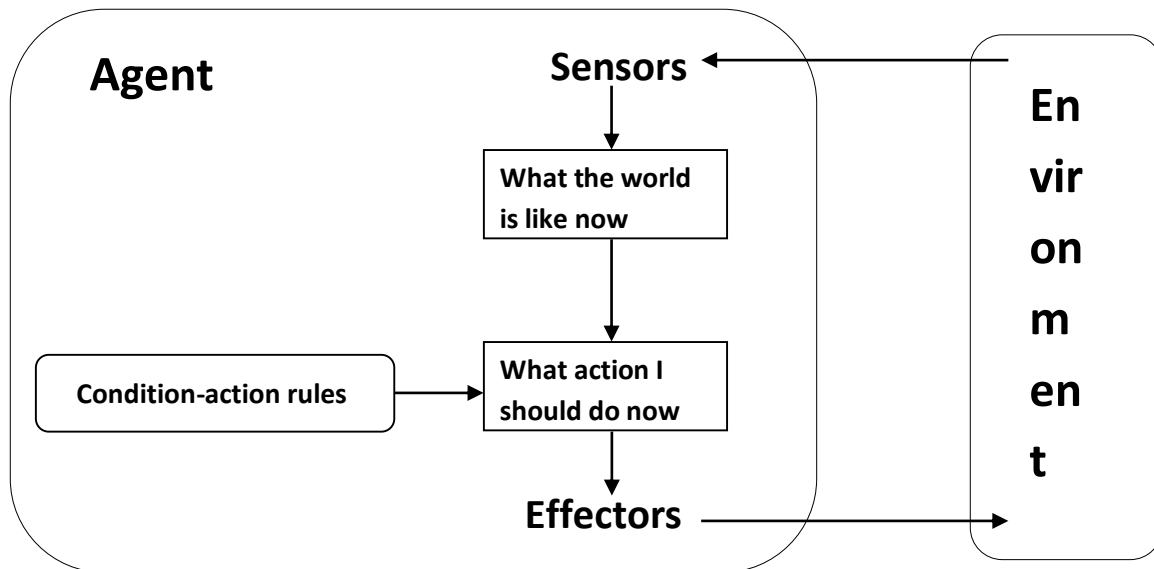
Actuators: Steering, accelerating, brake, horn, speaker/display,...

Sensors: Video, sonar, speedometer, engine sensors, keyboard, GPS, ...

Types of Agent

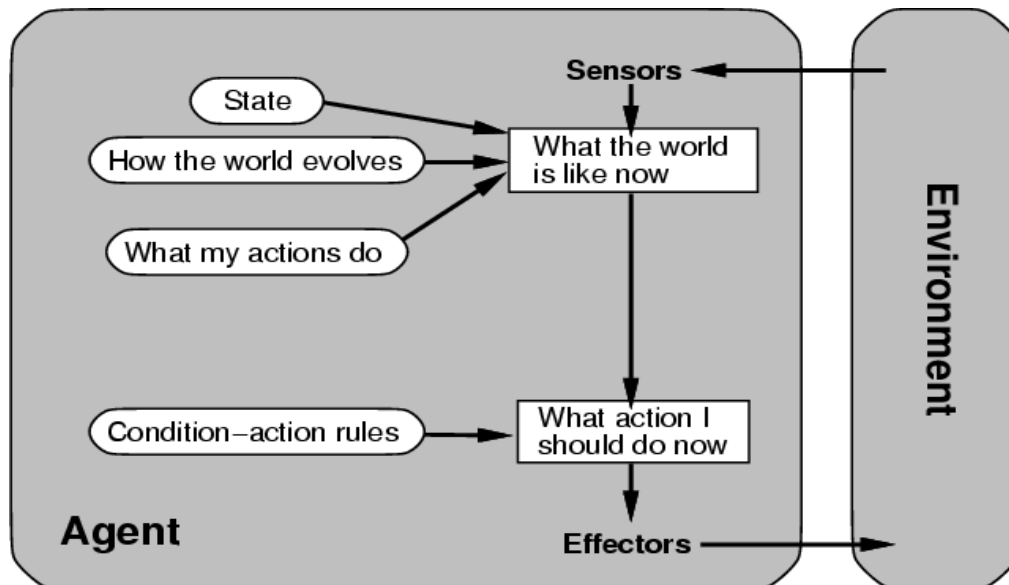
Simple Reflex Agent

- **Table lookup** of percept- action pairs defining all possible condition- action rules necessary to interact in an environment
- **Problems**
 - Too big to generate and to store (Chess has about 10^{120} states, for example)
 - No knowledge of non- perceptual parts of the current state
 - Not adaptive to changes in the environment; requires entire table to be updated if changes occur
- Use *condition-action* rules to summarize portions of the table



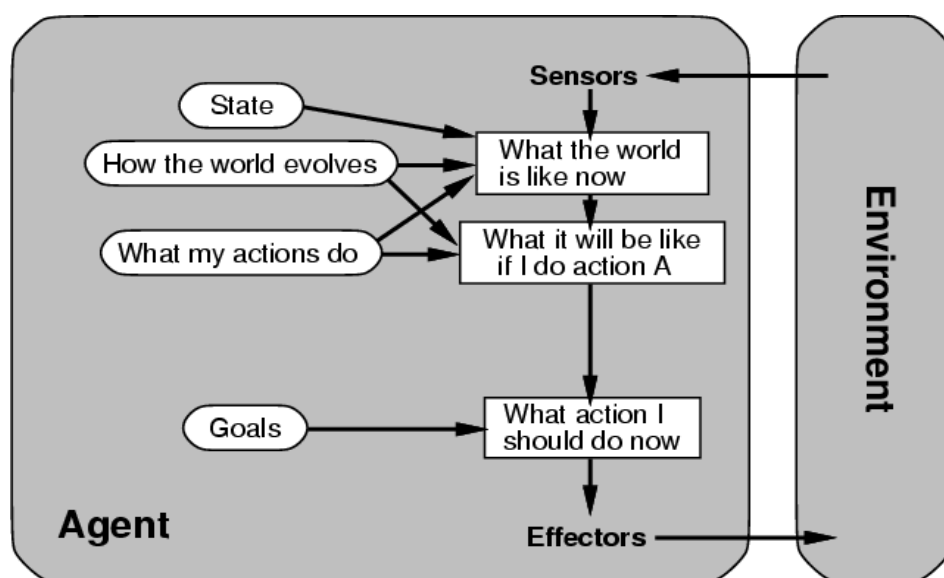
Model Based Agents (Reflex Agent with Internal State)

- Encode "internal state" of the world to remember the past as contained in earlier percepts
- Needed because sensors do not usually give the entire state of the world at each input, so perception of the environment is captured over time. "State" used to encode different "world states" that generate the same immediate percept.
- Requires ability to represent change in the world; one possibility is to represent just the latest state, but then can't reason about hypothetical courses of action



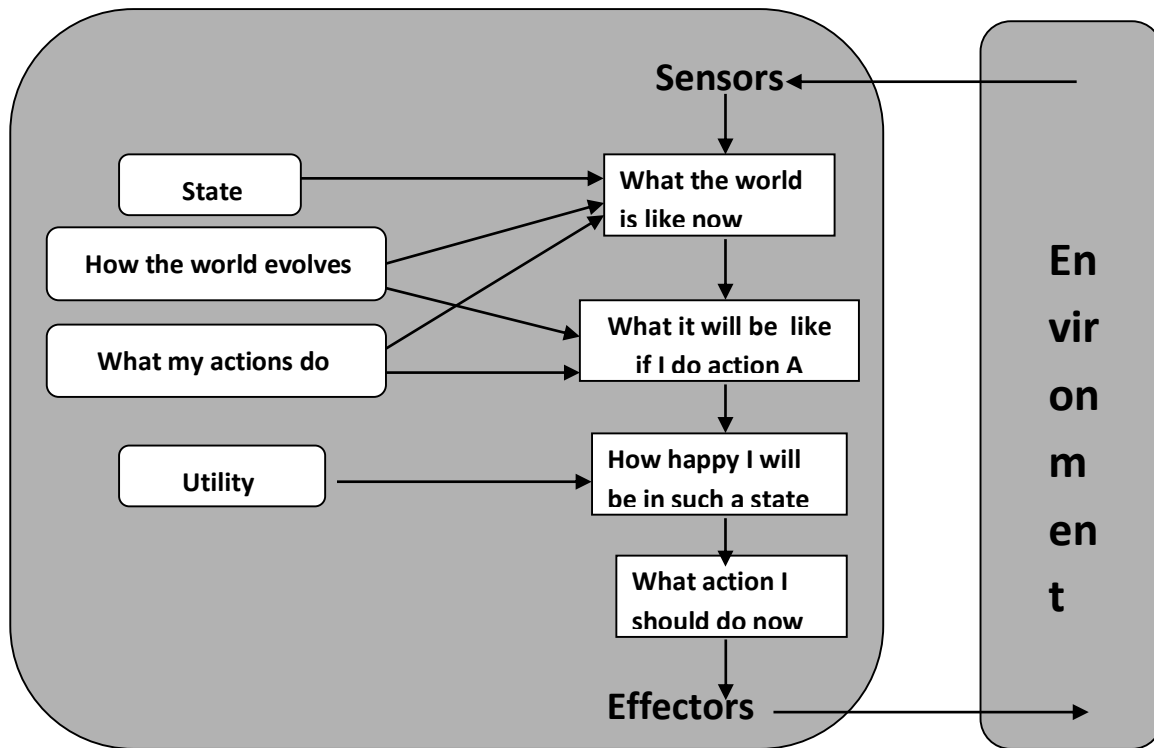
Goal- Based Agent

- Choose actions so as to achieve a (given or computed) goal.
- A goal is a description of a desirable situation
- Keeping track of the current state is often not enough -- need to add goals to decide which situations are good
- Deliberative instead of reactive
- May have to consider long sequences of possible actions before deciding if goal is achieved -- involves consideration of the future, "what will happen if I do...?"



Utility- Based Agent

- When there are multiple possible alternatives, how to decide which one is best?
- A goal specifies a crude distinction between a happy and unhappy state, but often need a more general performance measure that describes "degree of happiness"
- Utility function **U: States --> Reals** indicating a measure of success or happiness when at a given state
- Allows decisions comparing choice between conflicting goals, and choice between likelihood of success and importance of goal (if achievement is uncertain)



Learning Agent:

Refer Book: AI by Russel and Norvig

[Unit 3: Problem Solving]
Artificial Intelligence (CSC 355)

Jagadish Bhatta
Central Department of Computer Science & Information Technology
Tribhuvan University

Problem Solving:

Problem solving, particularly in artificial intelligence, may be characterized as a systematic search through a range of possible actions in order to reach some predefined goal or solution. Problem-solving methods divide into special purpose and general purpose. A special-purpose method is tailor-made for a particular problem and often exploits very specific features of the situation in which the problem is embedded. In contrast, a general-purpose method is applicable to a wide variety of problems. One general-purpose technique used in AI is means-end analysis—a step-by-step, or incremental, reduction of the difference between the current state and the final goal.

Four general steps in problem solving:

- Goal formulation
 - What are the successful world states
- Problem formulation
 - What actions and states to consider given the goal
- Search
 - Determine the possible sequence of actions that lead to the states of known values and then choosing the best sequence.
- Execute
 - Give the solution perform the actions.

Problem formulation:

A problem is defined by:

- An initial state: State from which agent start
- Successor function: Description of possible actions available to the agent.
- Goal test: Determine whether the given state is goal state or not
- Path cost: Sum of cost of each path from initial state to the given state.

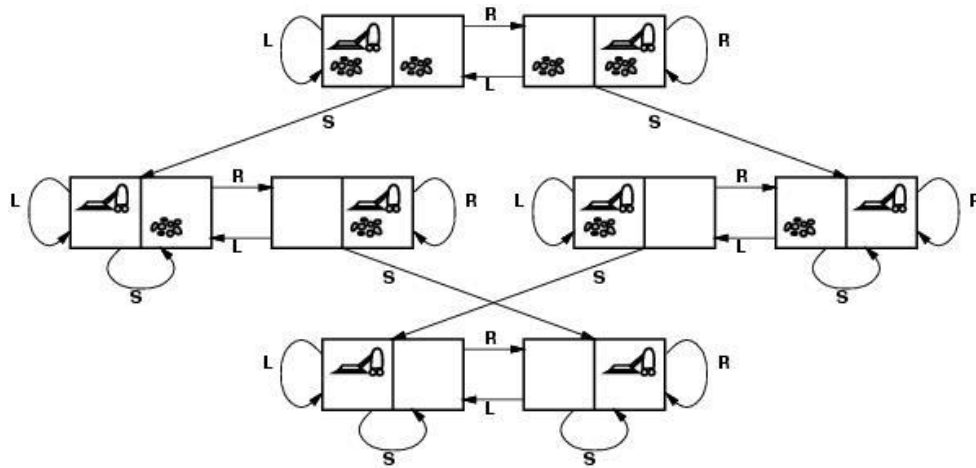
A solution is a sequence of actions from initial to goal state. Optimal solution has the lowest path cost.

State Space representation

The state space is commonly defined as a directed graph in which each node is a state and each arc represents the application of an operator transforming a state to a successor state.

A **solution** is a path from the initial state to a goal state.

State Space representation of Vacuum World Problem:



States?? two locations with or without dirt: $2 \times 2^2=8$ states.

Initial state?? Any state can be initial

Actions?? {Left, Right, Suck}

Goal test?? Check whether squares are clean.

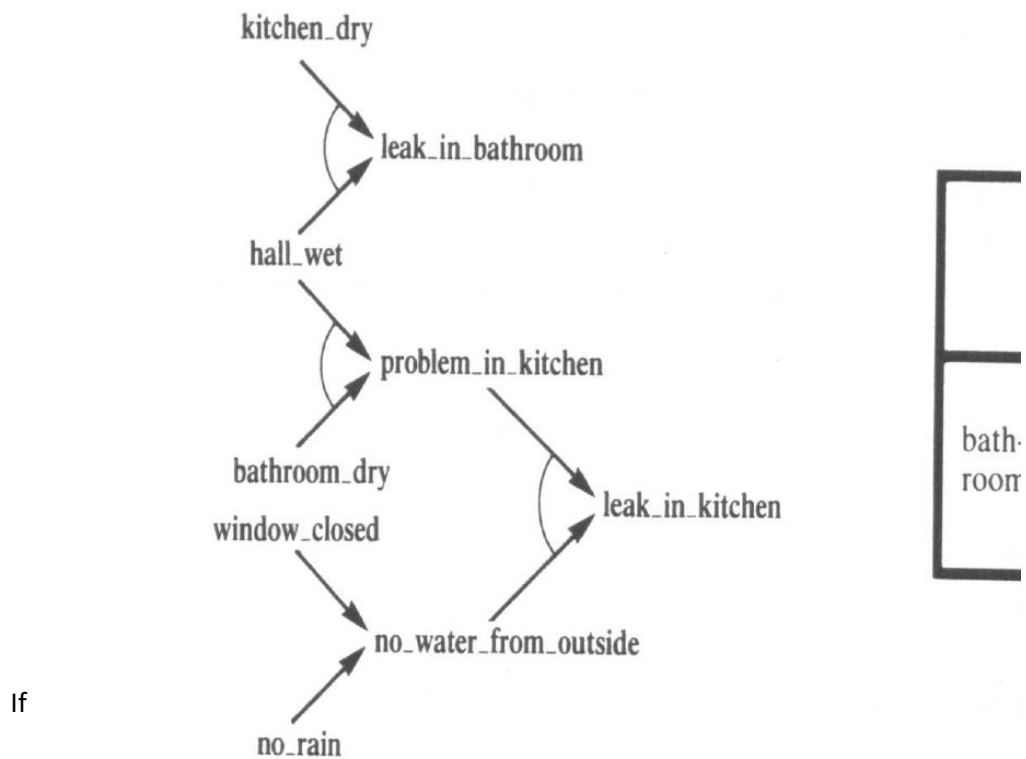
Path cost?? Number of actions to reach goal.

For following topics refer Russell and Norvig's Chapter 3 from pages 87-96.

Problem Types: Toy Problems & Real World Problems (Discussed in class) .

Well Defined Problems (Discussed in class).

Water Leakage Problem:



If

hall_wet and kitchen_dry

then

leak_in_bathroom

If

hall_wet and bathroom_dry

then

problem_in_kitchen

If

window_closed or no_rain

then

no_water_from_outside

Production System:

A **production system** (or **production rule system**) is a computer program typically used to provide some form of artificial intelligence, which consists primarily of a set of rules about behavior. These rules, termed **productions**, are a basic representation found useful in automated planning, expert systems and action selection. A production system provides the mechanism necessary to execute productions in order to achieve some goal for the system.

Productions consist of two parts: a sensory precondition (or "IF" statement) and an action (or "THEN"). If a production's precondition matches the current state of the world, then the production is said to be *triggered*. If a production's action is executed, it is said to have *fired*. A production system also contains a database, sometimes called working memory, which maintains data about current state or knowledge, and a rule interpreter. The rule interpreter must provide a mechanism for prioritizing productions when more than one is triggered.

The underlying idea of production systems is to represent knowledge in the form of condition-action pairs called production rules:

If the condition C is satisfied then the action A is appropriate.

Types of production rules

Situation-action rules

If it is raining then open the umbrella.

Inference rules

If Cesar is a man then Cesar is a person

Production system is also called *ruled-based system*

Architecture of Production System:

Short Term Memory:

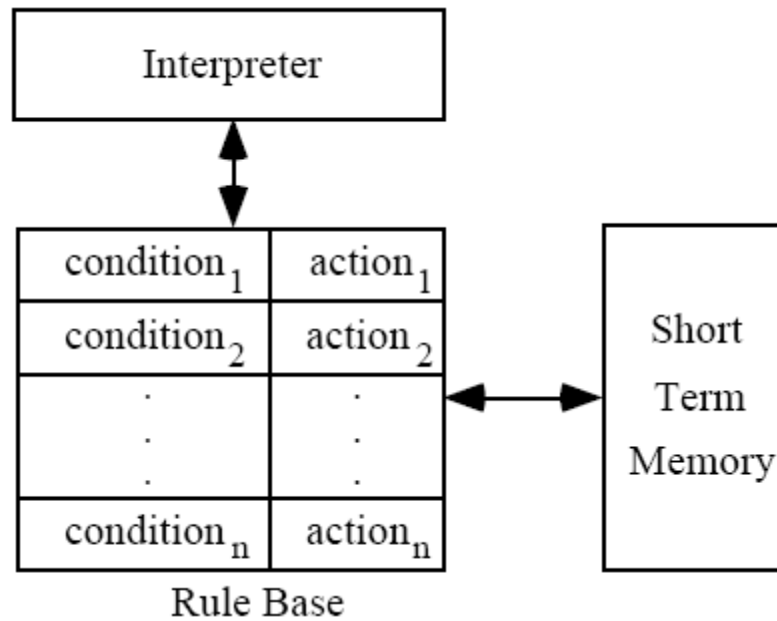
- Contains the description of the current state.

Set of Production Rules:

- Set of condition-action pairs and defines a single chunk of problem solving knowledge.

Interpreter:

- A mechanism to examine the short term memory and to determine which rules to fire (According to some strategies such as DFS, BFS, Priority, first-encounter etc)



The execution of a production system can be defined as a series of recognize-act cycles:
Match –memory contain matched against condition of production rules, this produces a subset of production called **conflict set**. **Conflict resolution** –one of the production in the conflict set is then selected, **Apply the rule**.

Consider an example:

Problem: Sorting a string composed of letters a, b & c.

Short Term Memory: cbaca

Production Set:

1. $ba \rightarrow ab$
2. $ca \rightarrow ac$
3. $cb \rightarrow bc$

Interpreter: Choose one rule according to some strategy.

Iteration #	Memory	Conflict Set	Rule fired
0	cbaca	1, 2, 3	1
1	cabca	2	2
2	acbca	2, 3	2
3	acbac	1, 3	1
4	acabc	2	2
5	aacbc	3	3
6	aabcc	0	halt

Production System: The water jug problem

Problem:

There are two jugs, a 4-gallon one and a 3-gallon one. Neither jug has any measuring markers on it. There is a pump that can be used to fill the jugs with water.

How can you get exactly n (0, 1, 2, 3, 4) gallons of water into one of the two jugs ?

Solution Paradigm:

- build a simple production system for solving this problem.
- represent the problem by using the state space paradigm.

State = (x, y) ; where: x represents the number of gallons in the 4-gallon jug; y represents the number of gallons in the 3-gallon jug. $x \in \{0, 1, 2, 3, 4\}$ and $y \in \{0, 1, 2, 3\}$.

The initial state represents the initial content of the two jugs.

For instance, it may be $(2, 3)$, meaning that the 4-gallon jug contains 2 gallons of water and the 3-gallon jug contains three gallons of water.

The goal state is the desired content of the two jugs.

The left hand side of a production rule indicates the state in which the rule is applicable and the right hand side indicates the state resulting after the application of the rule.

For instance;

- (x, y) such that $x < 4 \rightarrow (4, y)$ represents the production
If the 4-gallon jug is not full then fill it from the pump.

The rule base contains the following production rules:

1. (x, y) such that $x < 4 \rightarrow (4, y)$; Fill the 4-gallon jug from pump
2. (x, y) such that $y < 3 \rightarrow (x, 3)$; Fill the 3-gallon jug from pump
3. (x, y) such that $x > 0 \rightarrow (0, y)$; Empty the 4-gallon jug on the ground
4. (x, y) such that $y > 0 \rightarrow (x, 0)$; Empty the 3-gallon jug on the ground
5. (x, y) such that $x + y \geq 4, x < 4, y > 0 \rightarrow (4, y - (4 - x))$
; Completely fill the 4-gallon jug from the 3-gallon jug
6. (x, y) such that $x + y \geq 3, x > 0, y < 3 \rightarrow (x - (3 - y), 3)$
; Completely fill the 3-gallon jug from the 4-gallon jug
7. (x, y) such that $x + y \leq 4, y > 0 \rightarrow (x+y, 0)$
; Empty the 3-gallon jug into the 4-gallon jug
8. (x, y) such that $x + y \leq 3, x > 0 \rightarrow (0, x + y)$
; Empty the 4-gallon jug into the 3-gallon jug

The short term memory contains the current state (x, y) .

Let us consider the initial situation (0, 0) and the goal situation (n, 2)

short term memory : (0, 0)

1. Match: 1, 2. 2. Conflict resolution: select rule 2. 3. Apply the rule

short term memory becomes (0, 3)

1. Match: 1, 4, 7 2. Conflict resolution: select rule 7 3. Apply the rule

short term memory becomes (3, 0)

1. Match: 1, 2, 3, 6, 8 2. Conflict resolution: select rule 2 3. Apply the rule

short term memory becomes (3, 3)

1. Match: 1, 3, 4, 5 2. Conflict resolution: select rule 5 3. Apply the rule

short term memory becomes (4, 2) Goal achieved

The sequence of the applied rules:

Fill the 3-gallon jug from pump

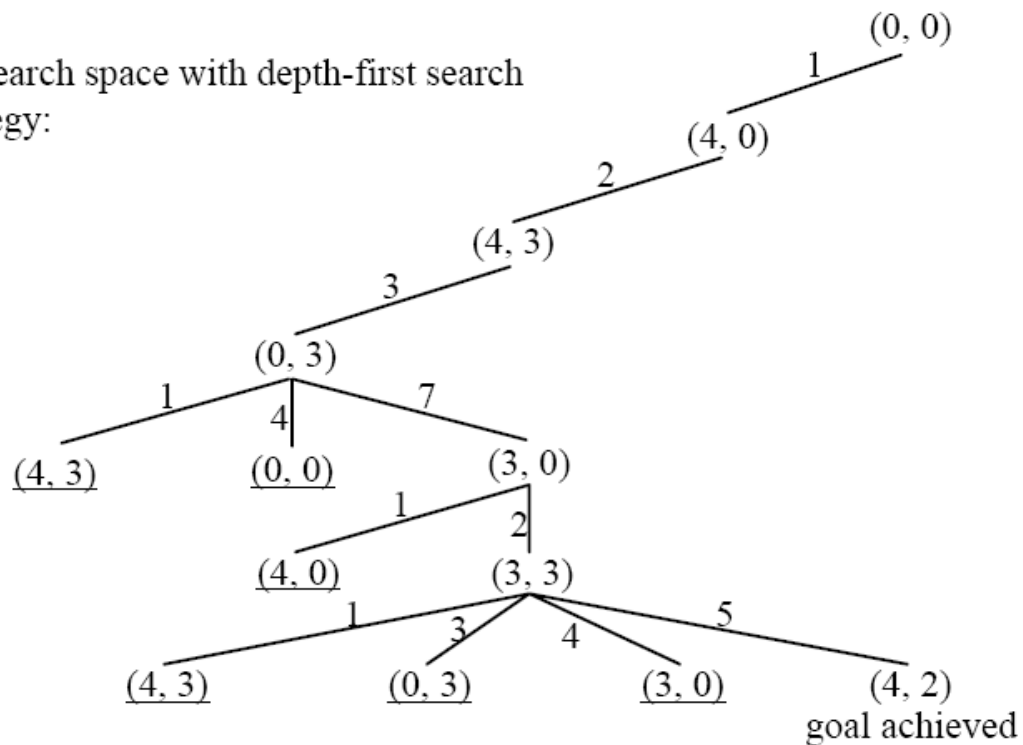
Empty the 3-gallon jug into the 4-gallon jug

Fill the 3-gallon jug from pump

Fill the 4-gallon jug from the 3-gallon jug

The Water Jug Problem: Representation

the search space with depth-first search strategy:



Constraint Satisfaction Problem:

A **Constraint Satisfaction Problem** is characterized by:

- a set of variables $\{x_1, x_2, \dots, x_n\}$,
- for each variable x_i a *domain* D_i with the possible values for that variable, and
- a set of *constraints*, i.e. relations, that are assumed to hold between the values of the variables. [These relations can be given intentionally, i.e. as a formula, or extensionally, i.e. as a set, or procedurally, i.e. with an appropriate generating or recognizing function.] We will only consider constraints involving one or two variables.

The constraint satisfaction problem is to find, for each i from 1 to n , a value in D_i for x_i so that all constraints are satisfied. Means that, we must find a value for each of the variables that satisfies all of the constraints.

A CS problem can easily be stated as a sentence in first order logic, of the form:

$$(\text{exist } x_1) \dots (\text{exist } x_n) (D_1(x_1) \ \& \ \dots \ D_n(x_n) \Rightarrow C_1 \dots C_m)$$

A CS problem is usually represented as an undirected graph, called *Constraint Graph* where the nodes are the variables and the edges are the binary constraints. Unary constraints can be disposed of by just redefining the domains to contain only the values that satisfy all the unary constraints. Higher order constraints are represented by hyperarcs. In the following we restrict our attention to the case of unary and binary constraints.

Formally, a constraint satisfaction problem is defined as a triple $\langle X, D, C \rangle$, where X is a set of variables, D is a domain of values, and C is a set of constraints. Every constraint is in turn a pair $\langle t, R \rangle$, where t is a tuple of variables and R is a set of tuples of values; all these tuples having the same number of elements; as a result R is a relation. An evaluation of the variables is a function from variables to values, $v : X \rightarrow D$. Such an evaluation satisfies a constraint $\langle (x_1, \dots, x_n), R \rangle$ if $(v(x_1), \dots, v(x_n)) \in R$. A solution is an evaluation that satisfies all constraints.

Constraints

- A constraint is a relation between a **local** collection of variables.
- The constraint restricts the values that these variables can simultaneously have.
- For example, **all-diff(X1, X2, X3)**. This constraint says that X1, X2, and X3 must take on different values. Say that $\{1,2,3\}$ is the set of values for each of these variables then:

X1=1, X2=2, X3=3 OK X1=1, X2=1, X3=3 NO

The constraints are the key component in expressing a problem as a CSP.

- The constraints are determined by how the variables and the set of values are chosen.
- Each constraint consists of;
 1. A set of variables it is over.
 2. A specification of the sets of assignments to those variables that satisfy the constraint.

- The idea is that we break the problem up into a set of distinct conditions each of which have to be satisfied for the problem to be solved.

Example: In N-Queens: Place N queens on an NxN chess board so that no queen can attack any other queen.

- No queen can attack any other queen.
- Given any two queens Q_i and Q_j they cannot attack each other.
- Now we translate each of these individual conditions into a separate constraint.
 - Q_i cannot attack $Q_j (i \neq j)$
 - Q_i is a queen to be placed in column i, Q_j is a queen to be placed in column j.
 - The value of Q_i and Q_j are the rows the queens are to be placed in.
- Note the translation is dependent on the representation we chose.
- **Queens can attack each other,**
 1. *Vertically*, if they are in the same column---this is impossible as Q_i and Q_j are placed in different columns.
 2. *Horizontally*, if they are in the same row---we need the constraint $Q_i \neq Q_j$.
 3. *Along a diagonal*, they cannot be the same number of columns apart as they are rows apart: we need the constraint $|i-j| \neq |Q_i-Q_j|$ ($| |$ is absolute value)
- **Representing the Constraints;**
 1. Between every pair of variables $(Q_i, Q_j) (i \neq j)$, we have a constraint C_{ij} .
 2. For each C_{ij} , an assignment of values to the variables $Q_i = A$ and $Q_j = B$, satisfies this constraint if and only if;
 - $A \neq B$
 - $|A-B| \neq |i-j|$
- **Solutions:**
 - A solution to the N-Queens problem will be any assignment of values to the variables Q_1, \dots, Q_N that satisfies all of the constraints.
 - Constraints can be over any collection of variables. In N-Queens we only need binary constraints---constraints over pairs of variables.

More Examples: Map Coloring Problem

[Unit 4: Searching]
Artificial Intelligence (CSC 355)

Jagadish Bhatta
Central Department of Computer Science & Information Technology
Tribhuvan University

Searching

A search problem

Figure below contains a representation of a map. The nodes represent cities, and the links represent direct road connections between cities. The number associated to a link represents the length of the corresponding road.

The search problem is to find a path from a city S to a city G

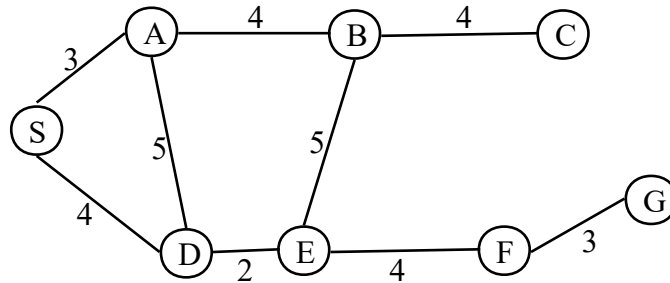


Figure : A graph representation of a map

This problem will be used to illustrate some search methods.

Search problems are part of a large number of real world applications:

- VLSI layout
- Path planning
- Robot navigation etc.

There are two broad classes of search methods:

- **uninformed (or blind) search methods;**
- **heuristically informed search methods.**

In the case of the uninformed search methods, the order in which potential solution paths are considered is arbitrary, using no domain-specific information to judge where the solution is likely to lie.

In the case of the heuristically informed search methods, one uses domain-dependent (heuristic) information in order to search the space more efficiently.

Measuring problem Solving Performance

We will evaluate the performance of a search algorithm in four ways

- **Completeness:** An algorithm is said to be complete if it definitely finds solution to the problem, if exist.

- **Time Complexity:** How long (worst or average case) does it take to find a solution?
Usually measured in terms of the **number of nodes expanded**
- **Space Complexity:** How much space is used by the algorithm? Usually measured in terms of the **maximum number of nodes in memory at a time**
- **Optimality/Admissibility:** If a solution is found, is it guaranteed to be an optimal one?
For example, is it the one with minimum cost?

Time and space complexity are measured in terms of

b -- maximum branching factor (number of successor of any node) of the search tree

m -- depth of the least-cost solution

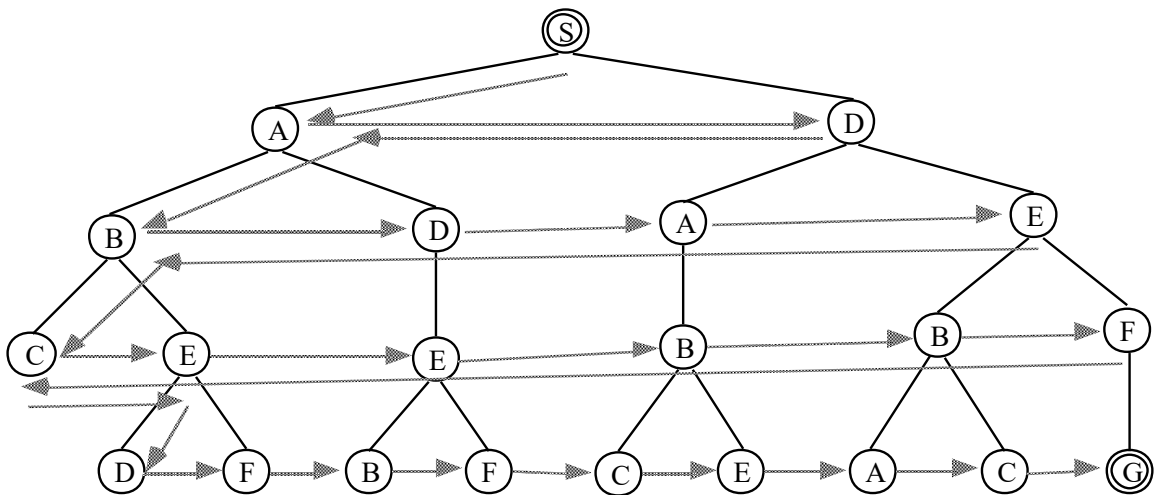
d -- maximum length of any path in the space

Breadth First Search

All nodes are expanded at a given depth in the search tree before any nodes at the next level are expanded until the goal reached.

Expand *shallowest* unexpanded node. *fringe* is implemented as a FIFO queue

Constraint: Do not generate as child node if the node is already parent to avoid more loop.



BFS Evaluation:

Completeness:

- Does it always find a solution if one exists?
- YES
- If shallowest goal node is at some finite depth d and If b is finite

Time complexity:

- Assume a state space where every state has b successors.
 - root has b successors, each node at the next level has again b successors (total b^2), ...
 - Assume solution is at depth d
 - Worst case; expand all except the last node at depth d

- Total no. of nodes generated:

$$b + b^2 + b^3 + \dots + b^d + (b^{d+1} - b) = O(b^{d+1})$$

Space complexity:

- Each node that is generated must remain in memory
- Total no. of nodes in memory:

$$1 + b + b^2 + b^3 + \dots + b^d + (b^{d+1} - b) = O(b^{d+1})$$

Optimal (i.e., admissible):

- if all paths have the same cost. Otherwise, not optimal but finds solution with shortest path length (shallowest solution). If each path does not have same path cost shallowest solution may not be optimal

Two lessons:

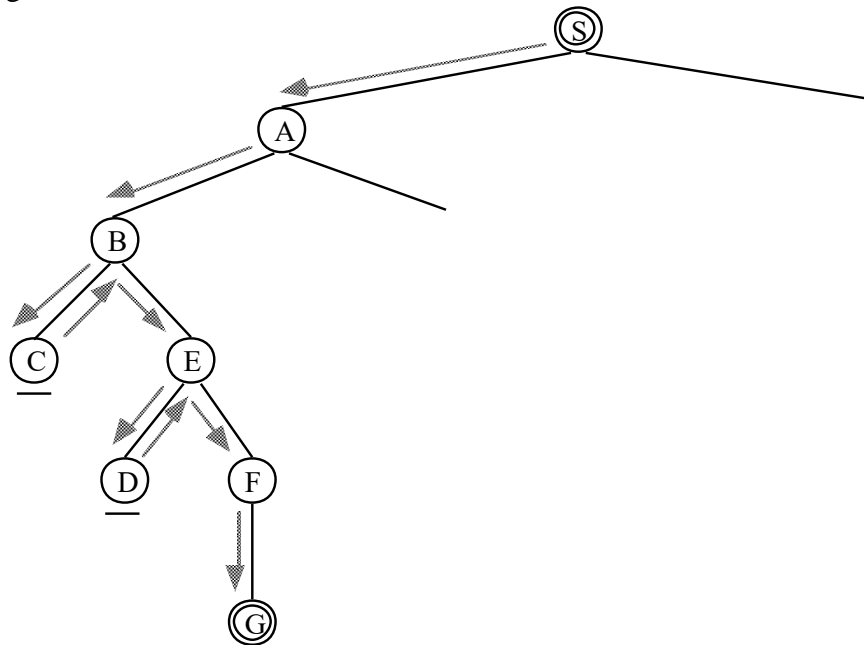
- Memory requirements are a bigger problem than its execution time.
- Exponential complexity search problems cannot be solved by uninformed search methods for any but the smallest instances.

DEPTH2	NODES	TIME	MEMORY
2	1100	0.11 seconds	1 megabyte
4	111100	11 seconds	106 megabytes
6	107	19 minutes	10 gigabytes
8	109	31 hours	1 terabyte
10	1011	129 days	101 terabytes
12	1013	35 years	10 petabytes
14	1015	3523 years	1 exabyte

Depth First Search

Looks for the goal node among all the children of the current node before using the sibling of this node i.e. **expand *deepest* unexpanded node.**

Fringe is implemented as a LIFO queue (=stack)



DFS Evaluation:

Completeness;

- Does it always find a solution if one exists?
- NO
 - If search space is infinite and search space contains loops then DFS may not find solution.

Time complexity;

- Let m is the maximum depth of the search tree. In the worst case Solution may exist at depth m .
- root has b successors, each node at the next level has again b successors (total b^2), ...
- Worst case; expand all except the last node at depth m
- Total no. of nodes generated:

$$b + b^2 + b^3 + \dots + b^m = O(b^m)$$

Space complexity:

- It needs to store only a single path from the root node to a leaf node, along with remaining unexpanded sibling nodes for each node on the path.
- Total no. of nodes in memory:

$$1 + b + b + b + \dots + b \text{ } m \text{ times} = O(bm)$$

Optimal (i.e., admissible):

- DFS expand deepest node first, if expands entire left sub-tree even if right sub-tree contains goal nodes at levels 2 or 3. Thus we can say DFS may not always give optimal solution.

Uniform Cost Search:

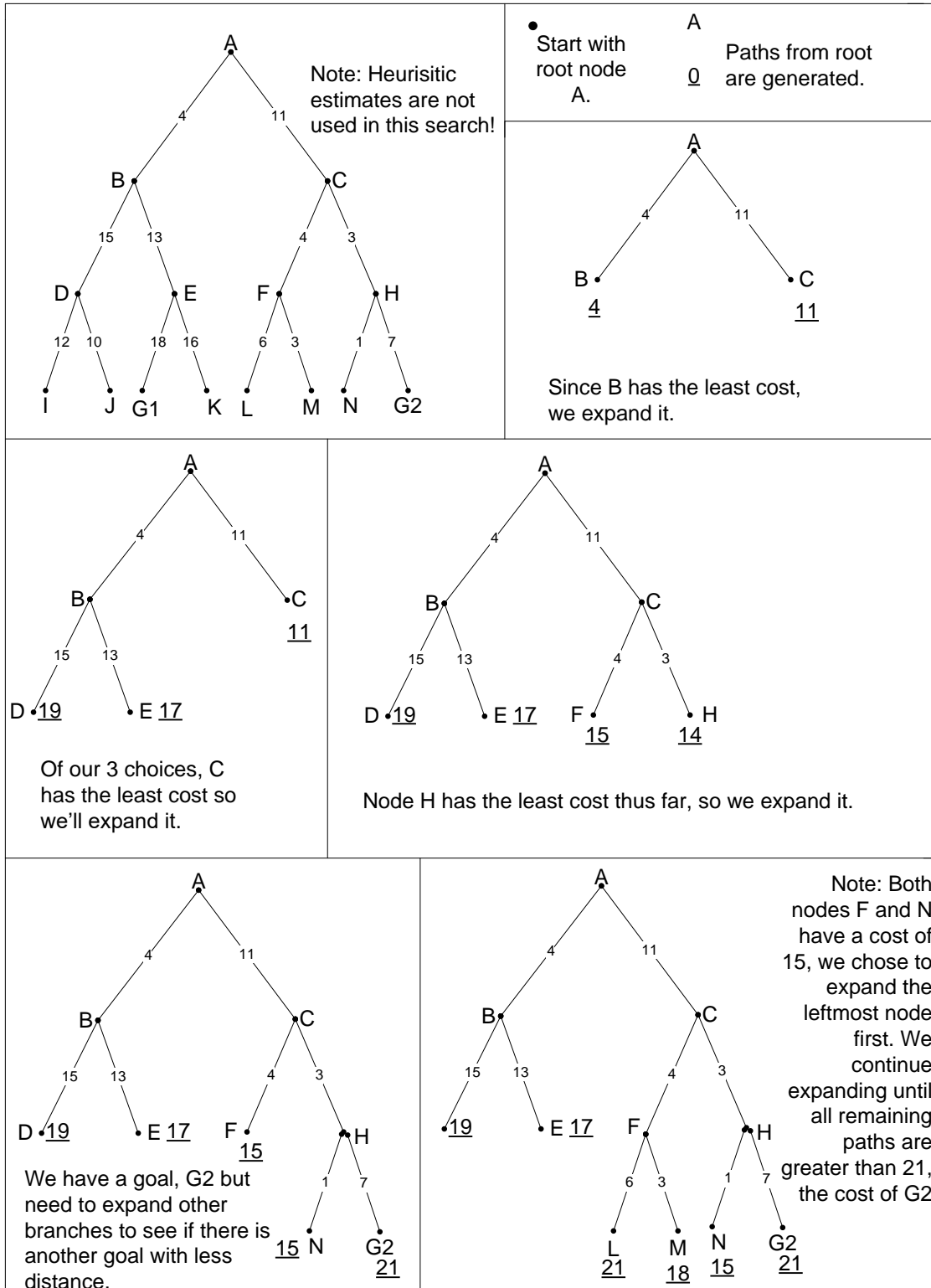
Uniform-cost search (UCS) is modified version of BFS to make optimal. It is basically a tree search algorithm used for traversing or searching a weighted tree, tree structure, or graph. The search begins at the root node. The search continues by visiting the next node which has the least total cost from the root. Nodes are visited in this manner until a goal state is reached.

Typically, the search algorithm involves expanding nodes by adding all unexpanded neighboring nodes that are connected by directed paths to a priority queue. In the queue, each node is associated with its total path cost from the root, where the least-cost paths are given highest priority. The node at the head of the queue is subsequently expanded, adding the next set of connected nodes with the total path cost from the root to the respective node. The uniform-cost search is **complete** and **optimal** if the cost of each step exceeds some positive bound ϵ .

Does not care about the number of steps, only care about total cost.

- Complete? Yes, if step cost $\geq \epsilon$ (small positive number).
- Time? Maximum as of BFS
- Space? Maximum as of BFS.
- Optimal? Yes

Consider an example:



Depth Limited Search:

The problem of unbounded trees can be solve by supplying depth-first search with a determined depth limit (nodes at depth are treated as they have no successors) –**Depth limited search. Depth-limited search** is an algorithm to explore the vertices of a graph. It is a modification of depth-first search and is used for example in the iterative deepening depth-first search algorithm.

Like the normal depth-first search, depth-limited search is an uninformed search. It works exactly like depth-first search, but avoids its drawbacks regarding completeness by imposing a maximum limit on the depth of the search. Even if the search could still expand a vertex beyond that depth, it will not do so and thereby it will not follow infinitely deep paths or get stuck in cycles. Therefore depth-limited search will find a solution if it is within the depth limit, which guarantees at least completeness on all graphs.

It solves the infinite-path problem of DFS. Yet it introduces another source of problem if we are unable to find good guess of l . Let d is the depth of shallowest solution.

If $l < d$ then incompleteness results.

If $l > d$ then not optimal.

Time complexity: $O(b^l)$

Space complexity: $O(b/l)$

Iterative Deepening Depth First Search:

In this strategy, depth-limited search is run repeatedly, increasing the depth limit with each iteration until it reaches d , the depth of the shallowest goal state. On each iteration, IDDFS visits the nodes in the search tree in the same order as depth-first search, but the cumulative order in which nodes are first visited, assuming no pruning, is effectively breadth-first.

IDDFS combines depth-first search's space-efficiency and breadth-first search's completeness (when the branching factor is finite). It is optimal when the path cost is a non-decreasing function of the depth of the node.

The technique of *iterative deepening* is based on this idea. *Iterative deepening* is depth-first search to a fixed depth in the tree being searched. If no solution is found up to this depth then the depth to be searched is increased and the whole 'bounded' depth-first search begun again.

It works by setting a depth of search -say, depth 1- and doing depth-first search to that depth. If a solution is found then the process stops -otherwise, increase the depth by, say, 1 and repeat until a solution is found. Note that every time we start up a new bounded depth search we *start from scratch* - i.e. we throw away any results from the previous search.

Now *iterative deepening* is a popular method of search. We explain why this is so.

Depth-first search can be implemented to be much cheaper than breadth-first search in terms of memory usage -but it is not guaranteed to find a solution even where one is guaranteed.

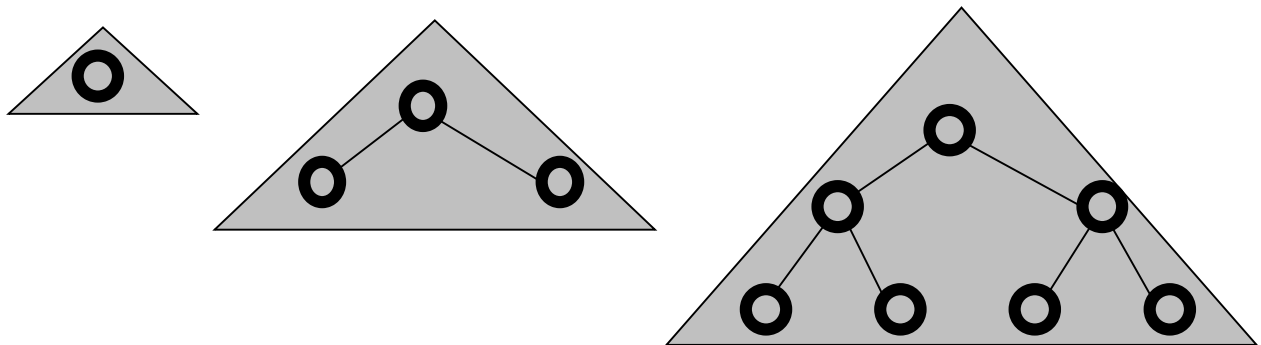
On the other hand, breadth-first search can be guaranteed to terminate if there is a winning state to be found and will always find the 'quickest' solution (in terms of how many steps need to be taken from the root node). It is, however, a very expensive method in terms of memory usage.

Iterative deepening is liked because it is an effective compromise between the two other methods of search. It is a form of depth-first search with a lower bound on how deep the search can go. Iterative deepening terminates if there is a solution. It can produce the same solution that breadth-first search would produce but does not require the same memory usage (as for breadth-first search).

Note that depth-first search achieves its efficiency by generating the next node to explore only when this needed. The breadth-first search algorithm has to grow all the search paths available until a solution is found -and this takes up memory. Iterative deepening achieves its memory saving in the same way that depth-first search does -at the expense of redoing some computations again and again (a time cost rather than a memory one). In the search illustrated, we had to visit node d three times in all!

- Complete (like BFS)
- Has linear memory requirements (like DFS)
- Classical time-space tradeoff.
- This is the preferred method for large state spaces, where the solution path length is unknown.

The overall idea goes as follows until the goal node is not found i.e. the depth limit is increased gradually.



Iterative Deepening search evaluation:

Artificial Intelligence

Completeness:

- YES (no infinite paths)

Time complexity:

- Algorithm seems costly due to repeated generation of certain states.
- Node generation:
 - level d: once
 - level d-1: 2
 - level d-2: 3
 - ...
 - level 2: d-1
 - level 1: d
- Total no. of nodes generated:
 $d.b + (d-1).b^2 + (d-2).b^3 + \dots + 1.b^d = O(b^d)$

Space complexity:

- It needs to store only a single path from the root node to a leaf node, along with remaining unexpanded sibling nodes for each node on the path.
- Total no. of nodes in memory:
 $1 + b + b + b + \dots + b \text{ } d \text{ times} = O(bd)$

Optimality:

- YES if path cost is non-decreasing function of the depth of the node.

Notice that BFS generates some nodes at depth $d+1$, whereas IDS does not. The result is that IDS is actually faster than BFS, despite the repeated generation of node.

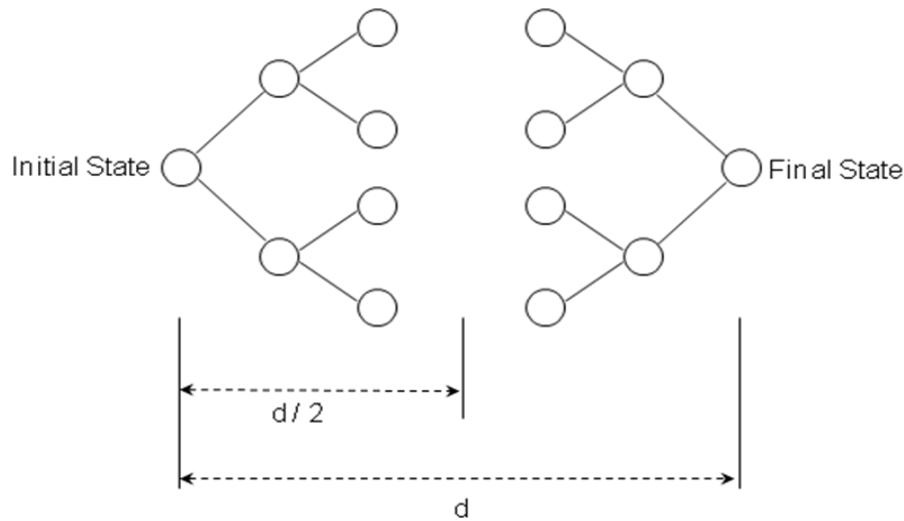
Example: Number of nodes generated for $b=10$ and $d=5$ solution at far right

$$N(\text{IDS}) = 50 + 400 + 3000 + 20000 + 100000 = 123450$$

$$N(\text{BFS}) = 10 + 100 + 1000 + 10000 + 100000 + 999990 = 1111100$$

Bidirectional Search:

This is a search algorithm which replaces a single search graph, which is likely to with two smaller graphs -- one starting from the initial state and one starting from the goal state. It then, expands nodes from the start and goal state simultaneously. Check at each stage if the nodes of one have been generated by the other, i.e, they meet in the middle. If so, the path concatenation is the solution.



- Completeness: yes
- Optimality: yes (If done with correct strategy- e.g. breadth first)
- Time complexity: $O(b^{d/2})$
- Space complexity: $O(b^{d/2})$

Problems: generate predecessors; many goal states; efficient check for node already visited by other half of the search; and, what kind of search.

Drawbacks of uniform search :

- Criterion to choose next node to expand depends only on a global criterion: level.
- Does not exploit the structure of the problem.
- One may prefer to use a more flexible rule that takes advantage of what is being discovered on the way, and hunches about what can be a good move.
- Very often, we can select which rule to apply by comparing the current state and the desired state

Heuristic Search:

Heuristic Search Uses domain-dependent (heuristic) information in order to search the space more efficiently.

Ways of using heuristic information:

- Deciding which node to expand next, instead of doing the expansion in a strictly breadth-first or depth-first order;
- In the course of expanding a node, deciding which successor or successors to generate, instead of blindly generating all possible successors at one time;
- Deciding that certain nodes should be discarded, or *pruned*, from the search space.

Heuristic Searches - Why Use?

- It may be too resource intensive (both time and space) to use a blind search
- Even if a blind search will work we may want a more efficient search method

Informed Search uses domain specific information to improve the search pattern

- Define a heuristic function, $h(n)$, that estimates the "goodness" of a node n .
- Specifically, $h(n)$ = estimated cost (or distance) of minimal cost path from n to a goal state.
- The heuristic function is an estimate, based on domain-specific information that is computable from the current state description, of how close we are to a goal.

Best-First Search

Idea: use an *evaluation function* $f(n)$ that gives an indication of which node to expand next for each node.

- usually gives an estimate to the goal.
- the node with the lowest value is expanded first.

A key component of $f(n)$ is a heuristic function, $h(n)$, which is a additional knowledge of the problem.

There is a whole family of best-first search strategies, each with a different evaluation function.

Typically, strategies use estimates of the cost of reaching the goal and try to minimize it.

Special cases: based on the evaluation function.

- Greedy best-first search
- A*search

Greedy Best First Search

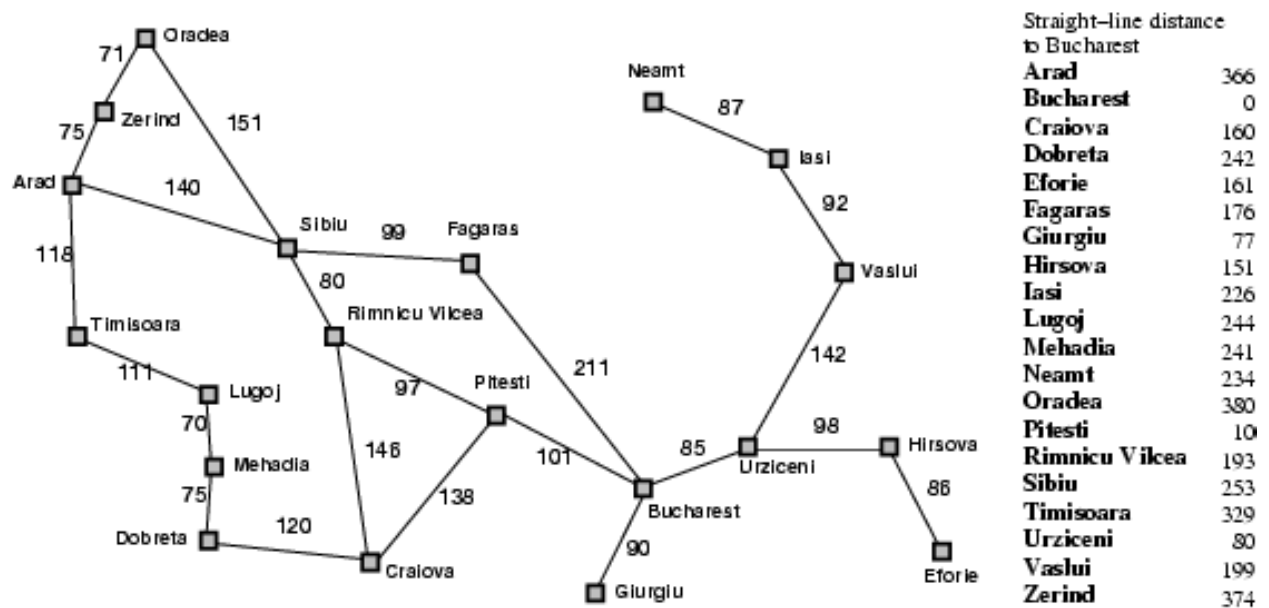
The best-first search part of the name means that it uses an evaluation function to select which node is to be expanded next. The node with the lowest evaluation is selected for expansion because that is the *best* node, since it supposedly has the closest path to the goal (if the heuristic is good). Unlike A* which uses both the link costs and a heuristic of the cost to the goal, greedy best-first search uses only the heuristic, and not any link costs. A disadvantage of this approach is that if the heuristic is not accurate, it can go down paths with high link cost since there might be a low heuristic for the connecting node.

Evaluation function $f(n) = h(n)$ (heuristic) = estimate of cost from n to *goal*.

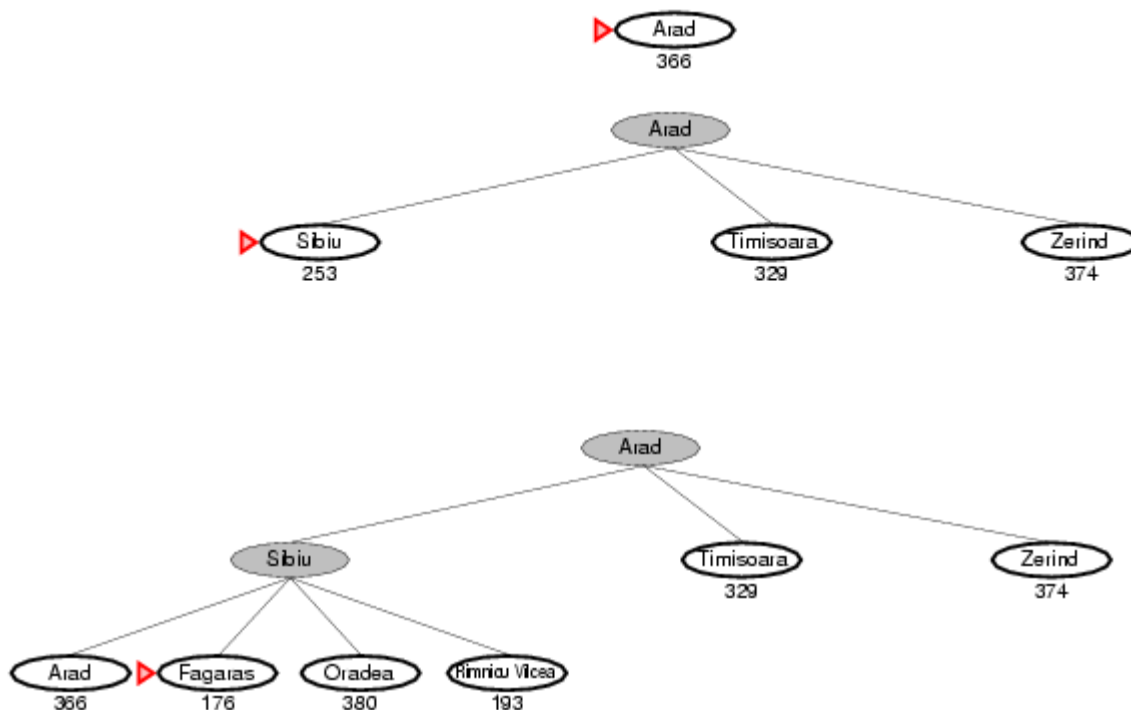
e.g., $h_{SLD}(n)$ = straight-line distance from n to goal

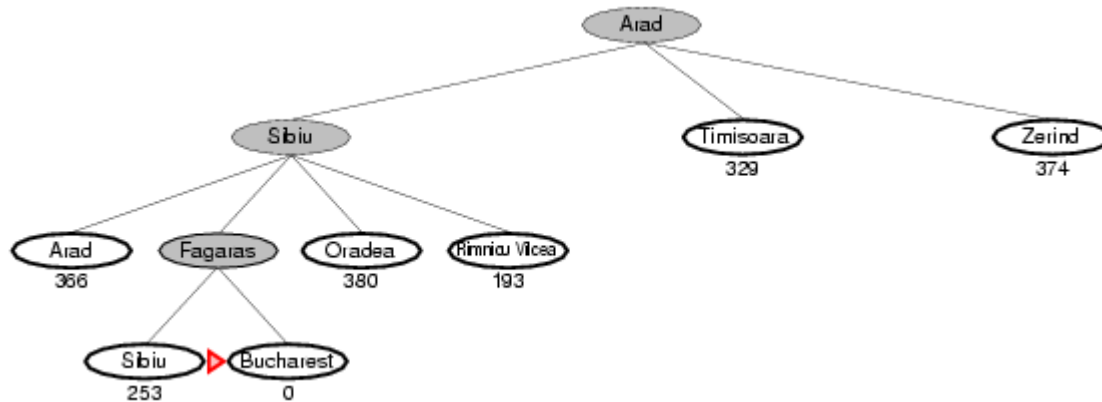
Greedy best-first search expands the node that appears to be closest to goal. The greedy best-first search algorithm is $O(b^m)$ in terms of space and time complexity. (Where b is the average branching factor (the average number of successors from a state), and m is the maximum depth of the search tree.)

Example: Given following graph of cities, starting at Arad city, problem is to reach to the Bucharest.



Solution using greedy best first can be as below:





A* Search : A Better Best-First Strategy

Greedy Best-first search

- minimizes estimated cost $h(n)$ from current node n to goal;
- is informed but (almost always) suboptimal and incomplete.

Admissible Heuristic:

A heuristic function is said to be **admissible** if it is no more than the lowest-cost path to the goal. In other words, a heuristic is admissible if it never overestimates the cost of reaching the goal. An admissible heuristic is also known as an **optimistic heuristic**.

An admissible heuristic is used to estimate the cost of reaching the goal state in an informed search algorithm. In order for a heuristic to be admissible to the search problem, the estimated cost must always be lower than the actual cost of reaching the goal state. The search algorithm uses the admissible heuristic to find an estimated optimal path to the goal state from the current node. For example, in A* search the evaluation function (where n is the current node) is: $f(n) = g(n) + h(n)$

where;

$f(n)$ = the evaluation function.

$g(n)$ = the cost from the start node to the current node

$h(n)$ = estimated cost from current node to goal.

$h(n)$ is calculated using the heuristic function. *With a non-admissible heuristic, the A* algorithm would overlook the optimal solution to a search problem due to an overestimation in $f(n)$.*

It is obvious that the SLD heuristic function is admissible as we can never find a shorter distance between any two towns.

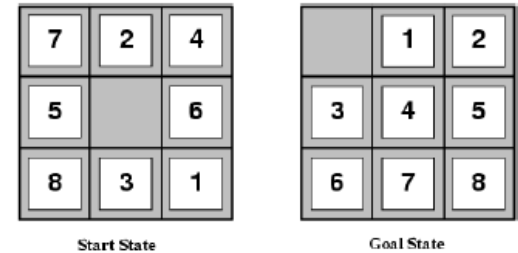
Formulating admissible heuristics:

- n is a node
- h is a heuristic
- $h(n)$ is cost indicated by h to reach a goal from n
- $C(n)$ is the actual cost to reach a goal from n
- h is admissible if

$$\forall n, h(n) \leq C(n)$$

For Example: 8-puzzle

Figure shows 8-puzzle start state and goal state. The solution is 26 steps long.



$h_1(n)$ = number of misplaced tiles

$h_2(n)$ = sum of the distance of the tiles from their goal position (not diagonal).

$h_1(S) = ?$ 8

$h_2(S) = ?$ 3+1+2+2+2+3+3+2 = 18

$h_n(S) = \max\{h_1(S), h_2(S)\} = 18$

Consistency (Monotonicity)

A heuristic is said to be consistent if for any node N and any successor N' of N , estimated cost to reach to the goal from node N is less than the sum of step cost from N to N' and estimated cost from node N' to goal node.

i.e $h(n) \leq c(n, n') + h(n')$

Where;

$h(n)$ = Estimated cost to reach to the goal node from node n

$c(n, n')$ = actual cost from n to n'

A* Search:

A* is a best first, informed graph search algorithm. A* is different from other best first search algorithms in that it uses a heuristic function $h(x)$ as well as the path cost to the node $g(x)$, in computing the cost $f(x) = h(x) + g(x)$ for the node. The $h(x)$ part of the $f(x)$ function must be an admissible heuristic; that is, it must not overestimate the distance to the goal. Thus, for an application like routing, $h(x)$ might represent the straight-line distance to the goal, since that is physically the smallest possible distance between any two points or nodes.

It finds a minimal cost-path joining the start node and a goal node for node n .

Evaluation function: $f(n) = g(n) + h(n)$

Artificial Intelligence

Where,

$g(n)$ = cost so far to reach n from root

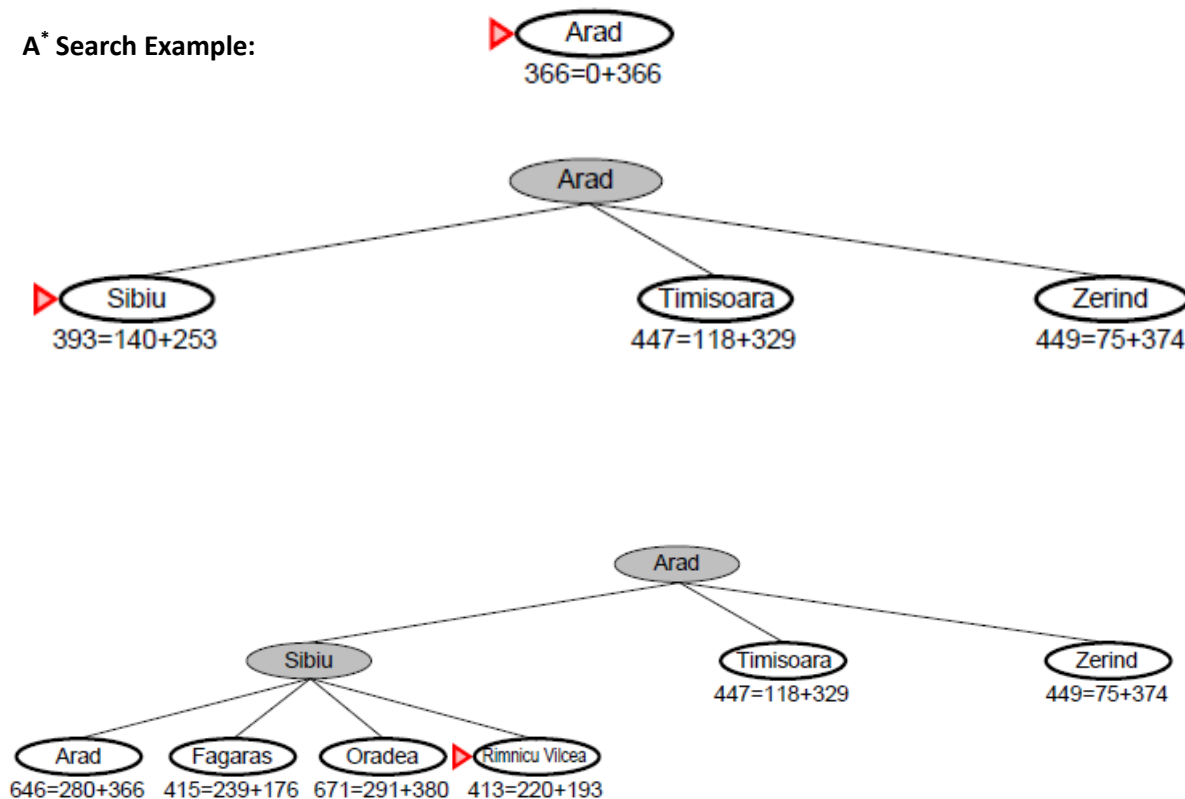
$h(n)$ = estimated cost to goal from n

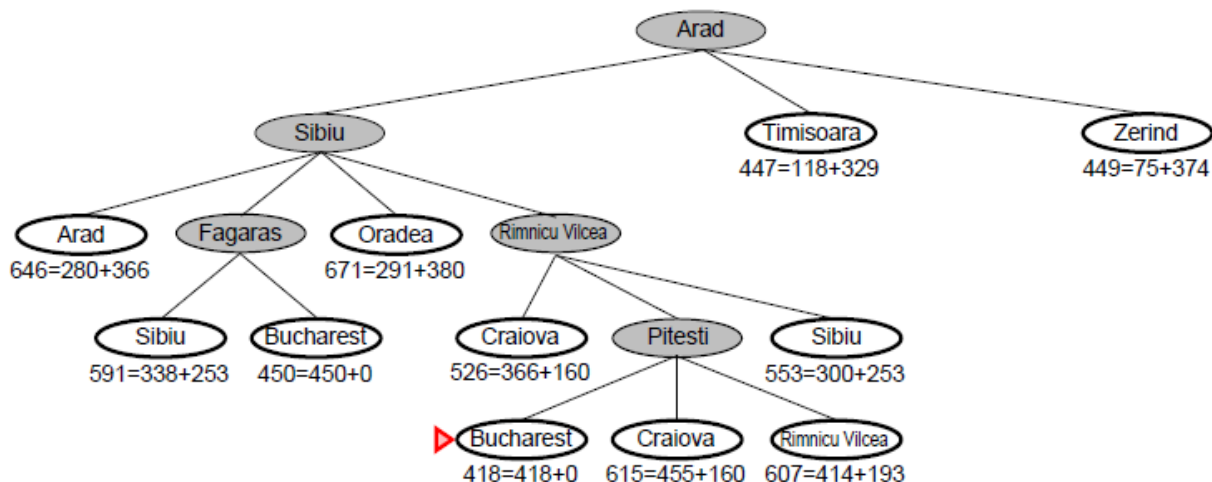
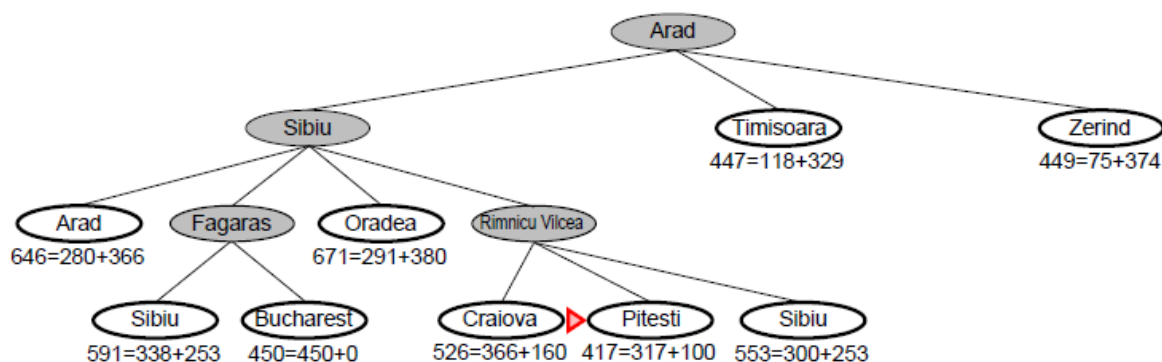
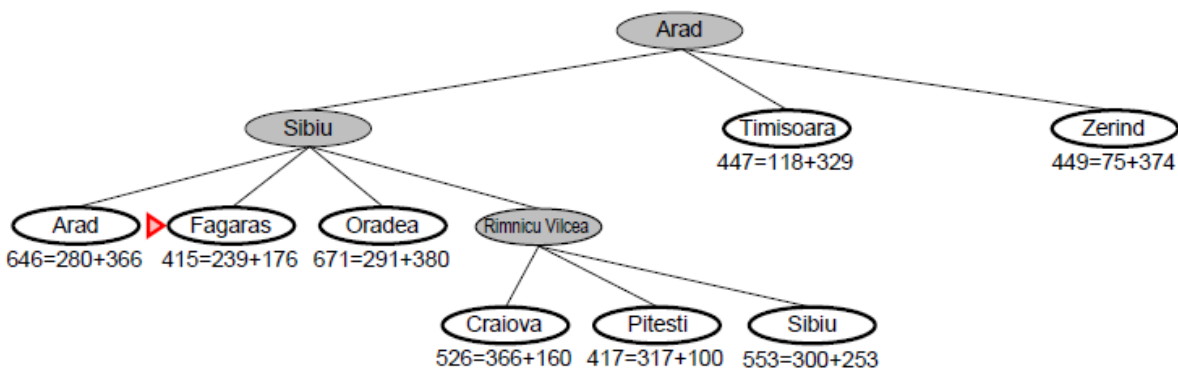
$f(n)$ = estimated total cost of path through n to goal

- combines the two by minimizing $f(n) = g(n) + h(n)$;
- is informed and, *under reasonable assumptions*, optimal and complete.

As A* traverses the graph, it follows a path of the lowest *known* path, keeping a sorted priority queue of alternate path segments along the way. If, at any point, a segment of the path being traversed has a higher cost than another encountered path segment, it abandons the higher-cost path segment and traverses the lower-cost path segment instead. This process continues until the goal is reached.

A* Search Example:





Admissibility and Optimality:

A* is admissible and considers fewer nodes than any other admissible search algorithm with the same heuristic. This is because A* uses an "optimistic" estimate of the cost of a path through every node that it considers—optimistic in that the true cost of a path through that node to the goal will be at least as great as the estimate. But, critically, as far as A* "knows", that optimistic estimate might be achievable.

Here is the main idea of the proof:

When A* terminates its search, it has found a path whose actual cost is lower than the estimated cost of any path through any open node. But since those estimates are optimistic, A* can safely ignore those nodes. In other words, A* will never overlook the possibility of a lower-cost path and so is admissible.

Suppose, now that some other search algorithm B terminates its search with a path whose actual cost is *not* less than the estimated cost of a path through some open node. Based on the heuristic information it has, Algorithm B cannot rule out the possibility that a path through that node has a lower cost. So while B might consider fewer nodes than A*, it cannot be admissible. Accordingly, A* considers the fewest nodes of any admissible search algorithm.

This is only true if both:

- A* uses an admissible heuristic. Otherwise, A* is not guaranteed to expand fewer nodes than another search algorithm with the same heuristic.
- A* solves only one search problem rather than a series of similar search problems. Otherwise, A* is not guaranteed to expand fewer nodes than incremental heuristic search algorithms

Thus, if estimated distance $h(n)$ never exceed the true distance $h^*(n)$ between the current node to goal node, the A* algorithm will always find a shortest path -This is known as the *admissibility* of A* algorithm and $h(n)$ is a admissible heuristic.

IF $0 \leq h(n) \leq h^*(n)$, and costs of all arcs are positive
THEN A* is guaranteed to find a solution path of minimal cost if any solution path exists.

Theorem: A* is optimal if $h(n)$ is admissible.

Suppose suboptimal goal G_2 in the queue.

Let n be an unexpanded node on a shortest path to optimal goal G and C^* be the cost of optimal goal node.

$$f(G_2) = h(G_2) + g(G_2)$$

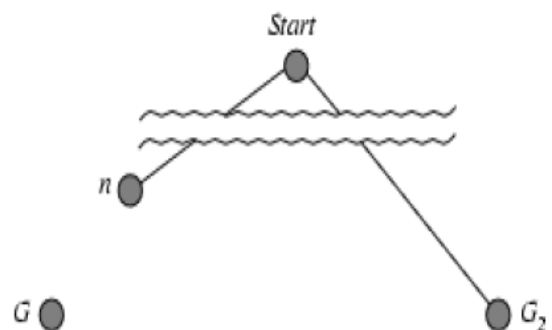
$$f(G_2) = g(G_2), \text{ since } h(G_2) = 0$$

$$f(G_2) > C^* \quad \dots\dots\dots(1)$$

Again, since $h(n)$ is admissible, It does not overestimates the cost of completing the solution path.

$$f(n) = g(n) + h(n) \leq C^* \quad \dots\dots\dots(2)$$

Now from (1) and (2) $f(n) \leq C^* < f(G_2)$



Since $f(G2) > f(n)$, A* will never select G2 for expansion. Thus A* gives us optimal solution when heuristic function is admissible.

Theorem: If $h(n)$ is consistent, then the values of $f(n)$ along the path are non-decreasing.

Suppose n' is successor of n , then

$$g(n') = g(n) + C(n, a, n')$$

We know that,

$$f(n') = g(n') + h(n')$$

$$f(n') = g(n) + C(n, a, n') + h(n') \dots\dots\dots(1)$$

A heuristic is consistent if

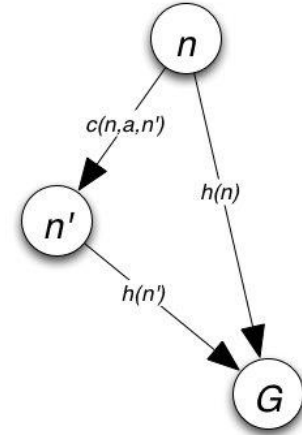
$$h(n) \leq C(n, a, n') + h(n') \dots\dots\dots(2)$$

Now from (1) and (2)

$$f(n') = g(n) + C(n, a, n') + h(n') \geq g(n) + h(n) = f(n)$$

$$f(n') \geq f(n)$$

$f(n)$ is non-decreasing along any path.



One more example: Maze Traversal (for A* Search)

Problem: To get from square A3 to square E2, one step at a time, avoiding obstacles (black squares).

Operators: (in order)

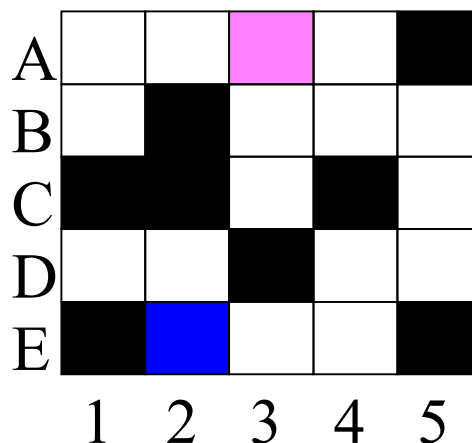
- go_left(n)
- go_down(n)
- go_right(n)

Each operator costs 1.

Heuristic: Manhattan distance

Start Position: A3

Goal: E2



$\frac{4(B4)}{h(B4)} = 2$
 $\frac{4(B4)}{h(B4)} = 3$
 $\frac{g(B1)}{h(B1)} = 3$
 $\frac{g(B1)}{h(B1)} = 4$
 $\frac{g(B5)}{h(B5)} = 3$
 $\frac{g(B5)}{h(B5)} = 6$

Hill climbing can be used to solve problems that have many solutions, some of which are better than others. **It starts with a random (potentially poor) solution, and iteratively makes small changes to the solution, each time improving it a little. When the algorithm cannot see any improvement anymore, it terminates.** Ideally, at that point the current solution is close to optimal, but it is not guaranteed that hill climbing will ever come close to the optimal solution.

Shiv Raj Pant

The algorithm starts with such a solution and makes small improvements to it, such as switching the order in which two cities are visited. Eventually, a much better route is obtained. In hill climbing the basic idea is to always head towards a state which is better than the current one. So, if you are at town A and you can get to town B and town C (and your target is town D) then you should make a move IF town B or C appear nearer to town D than town A does.

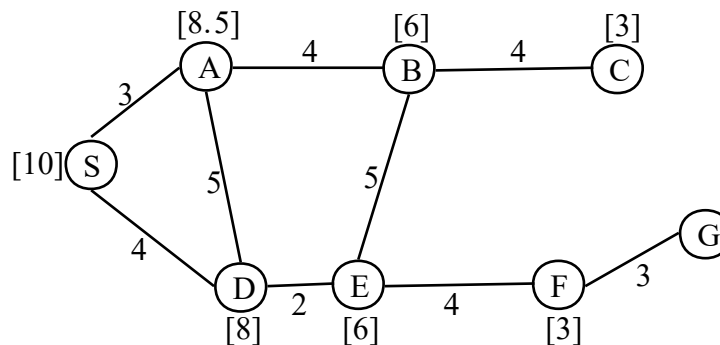
The hill climbing can be described as follows:

1. Start with *current-state* = initial-state.
2. Until *current-state* = goal-state OR there is no change in *current-state* do:
 - Get the successors of the current state and use the evaluation function to assign a score to each successor.
 - If one of the successors has a better score than the current-state then set the new current-state to be the successor with the best score.

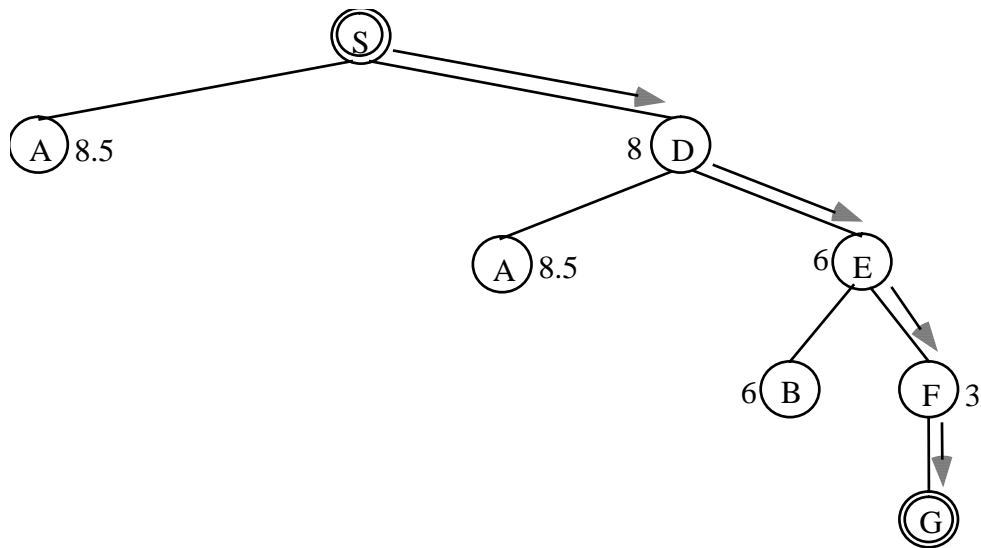
Hill climbing terminates when there are no successors of the current state which are better than the current state itself.

Hill climbing is depth-first search with a heuristic measurement that orders choices as nodes are expanded. It always selects the most promising successor of the node last expanded.

For instance, consider that the most promising successor of a node is the one that has the shortest straight-line distance to the goal node G. In figure below, the straight line distances between each city and goal G is indicated in square brackets, i.e. the heuristic.

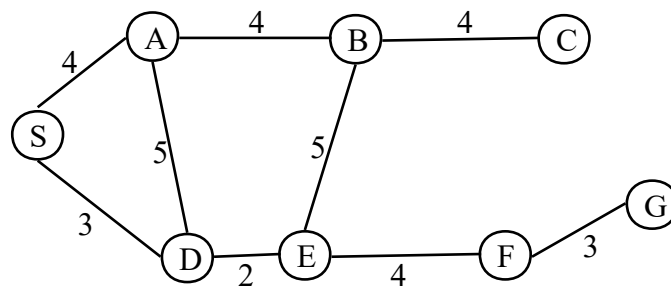


The hill climbing search from S to G proceeds as follows:



Exercise:

Apply the hill climbing algorithm to find a path from S to G, considering that the most promising successor of a node is its closest neighbor.



Note:

The difference between the hill climbing search method and the best first search method is the following one:

- the best first search method selects for expansion the most promising leaf node of the current search tree;
- the hill climbing search method selects for expansion the most promising successor of the node last expanded.

Problems with Hill Climbing

- Gets stuck at **local maxima** when we reach a position where there are no better neighbors, it is not a guarantee that we have found the best solution. **Ridge** is a sequence of local maxima.
- Another type of problem we may find with hill climbing searches is finding a **plateau**. This is an area where the search space is flat so that all neighbors return the same evaluation

Simulated Annealing:

It is motivated by the physical annealing process in which material is heated and slowly cooled into a uniform structure. Compared to hill climbing the main difference is that SA allows downwards steps. Simulated annealing also differs from hill climbing in that a move is selected at random and then decides whether to accept it. If the move is better than its current position then simulated annealing will always take it. If the move is worse (i.e. lesser quality) then it will be accepted based on some probability. The probability of accepting a worse state is given by the equation

$$P = \text{exponential}(-c / t) > r$$

Where

c	=	the change in the evaluation function
t	=	the current value
r	=	a random number between 0 and 1

The probability of accepting a worse state is a function of both the current value and the change in the cost function. The most common way of implementing an SA algorithm is to implement hill climbing with an accept function and modify it for SA

By analogy with this physical process, each step of the SA algorithm replaces the current solution by a random "nearby" solution, chosen with a probability that depends on the difference between the corresponding function values and on a global parameter T (called the *temperature*), that is gradually decreased during the process. The dependency is such that the current solution changes almost randomly when T is large, but increasingly "downhill" as T goes to zero. The allowance for "uphill" moves saves the method from becoming stuck at local optima—which are the bane of greedier methods.

Game Search:

Games are a form of *multi-agent environment*

- What do other agents do and how do they affect our success?
- Cooperative vs. competitive multi-agent environments.
- Competitive multi-agent environments give rise to adversarial search often known as *games*

Games – adversary

- Solution is strategy (strategy specifies move for every possible opponent reply).
- Time limits force an *approximate* solution
- Evaluation function: evaluate "goodness" of game position
- Examples: chess, checkers, Othello, backgammon

Difference between the search space of a game and the search space of a problem: In the first case it represents the moves of two (or more) players, whereas in the latter case it represents the "moves" of a single problem-solving agent.

An exemplary game: Tic-tac-toe

There are two players denoted by X and O. They are alternatively writing their letter in one of the 9 cells of a 3 by 3 board. The winner is the one who succeeds in writing three letters in line.

The game begins with an empty board. It ends in a win for one player and a loss for the other, or possibly in a draw.

A complete tree is a representation of all the possible plays of the game. The root node is the initial state, in which it is the first player's turn to move (the player X).

The successors of the initial state are the states the player can reach in one move, their successors are the states resulting from the other player's possible replies, and so on.

Terminal states are those representing a win for X, loss for X, or a draw.

Each path from the root node to a terminal node gives a different complete play of the game. Figure given below shows the initial search space of Tic-Tac-Toe.

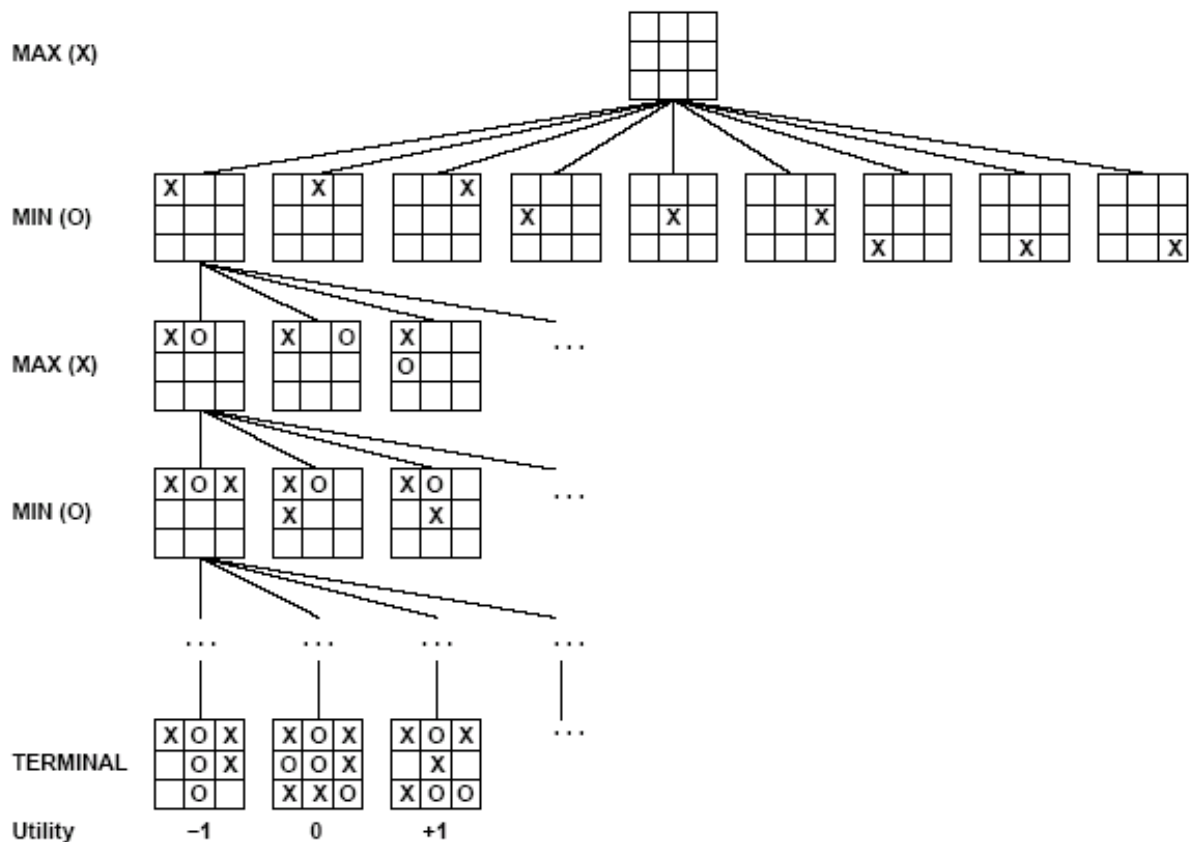


Fig: Partial game tree for Tic-Tac-Toe

A game can be formally defined as a kind of search problem as below:

- Initial state: It includes the board position and identifies the player to move.

- Successor function: It gives a list of (move, state) pairs each indicating a legal move and resulting state.
- Terminal test: This determines when the game is over. States where the game is ended are called terminal states.
- Utility function: It gives numerical value of terminal states. E.g. win (+1), loose (-1) and draw (0). Some games have a wider variety of possible outcomes eg. ranging from +92 to -192.

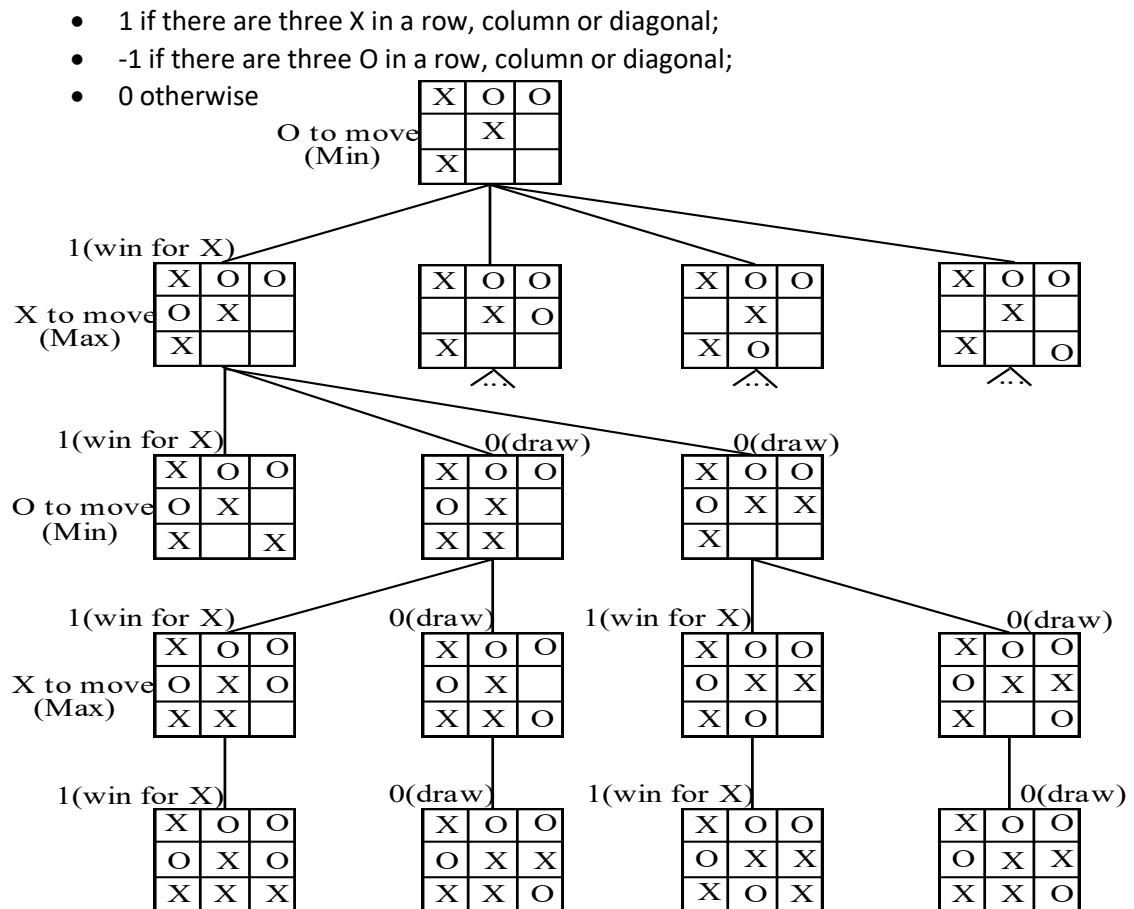
The Minimax Algorithm:

Let us assign the following values for the game: 1 for win by X, 0 for draw, -1 for loss by X.

Given the values of the terminal nodes (win for X (1), loss for X (-1), or draw (0)), the values of the non-terminal nodes are computed as follows:

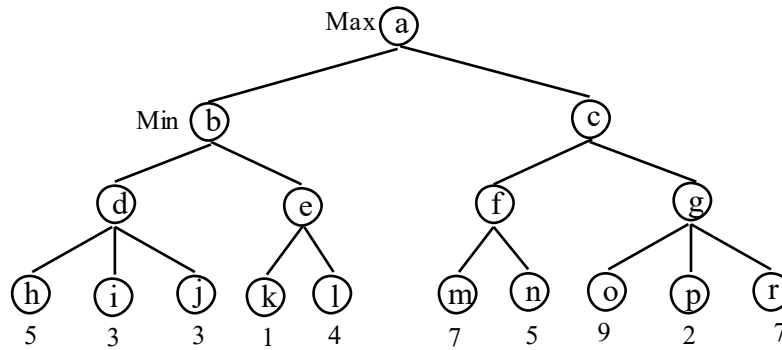
- the value of a node where it is the turn of player X to move is the maximum of the values of its successors (because X tries to maximize its outcome);
- the value of a node where it is the turn of player O to move is the minimum of the values of its successors (because O tries to minimize the outcome of X).

Figure below shows how the values of the nodes of the search tree are computed from the values of the leaves of the tree. The values of the leaves of the tree are given by the rules of the game:



An Example:

Consider the following game tree (drawn from the point of view of the Maximizing player):



Show what moves should be chosen by the two players, assuming that both are using the mini-max procedure.

Solution:

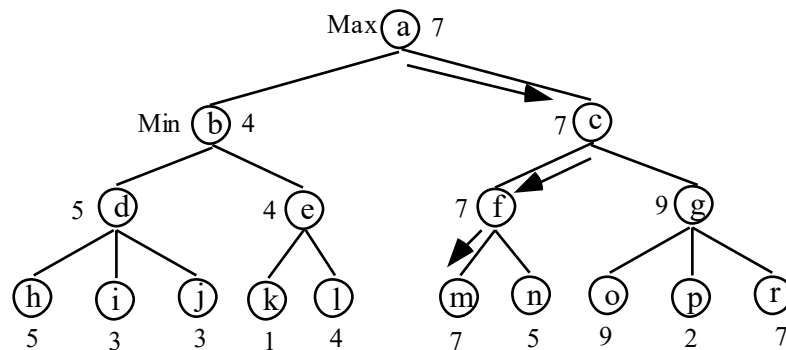


Figure 3.16: The mini-max path for the game tree

Alpha-Beta Pruning:

The problem with minimax search is that the number of game states it has to examine is exponential in the number of moves. Unfortunately, we can't eliminate the exponent, but we can effectively cut it in half. The idea is to compute the correct minimax decision without looking at every node in the game tree, which is the concept behind pruning. Here the idea is to eliminate large parts of the tree from consideration. The particular technique for pruning that we will discuss here is "**Alpha-Beta Pruning**". When this approach is applied to a standard minimax tree, it returns the same move as minimax would, but prunes away branches that cannot possibly influence the final decision. Alpha-

beta pruning can be applied to trees of any depth, and it is often possible to prune entire sub-trees rather than just leaves.

Alpha-beta pruning is a technique for evaluating nodes of a game tree that eliminates unnecessary evaluations. It uses two parameters, alpha and beta.

Alpha: is the value of the best (i.e. highest value) choice we have found so far at any choice point along the path for MAX.

Beta: is the value of the best (i.e. lowest-value) choice we have found so far at any choice point along the path for MIN.

Alpha-beta search updates the values of alpha and beta as it goes along and prunes the remaining branches at a node as soon as the value of the current node is known to be worse than the current alpha or beta for MAX or MIN respectively.

An alpha cutoff:

To apply this technique, one uses a parameter called alpha that represents a lower bound for the achievement of the Max player at a given node.

Let us consider that the current board situation corresponds to the node A in the following figure.

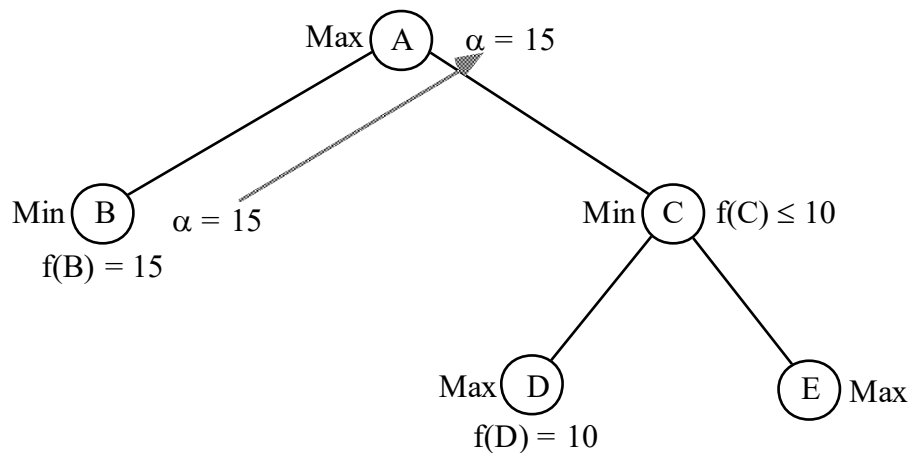


Figure 3.17: Illustration of the alpha cut-off.

The minimax method uses a depth-first search strategy in evaluating the descendants of a node. It will therefore estimate first the value of the node B. Let us suppose that this value has been evaluated to 15, either by using a static evaluation function, or by backing up from descendants omitted in the figure. If Max will move to B then it is guaranteed to achieve 15. Therefore 15 is a lower bound for the achievement of the Max player (it may still be possible to achieve more, depending on the values of the other descendants of A). Therefore, the value of α at node B is 15. This value is transmitted upward to the node A and will be used for evaluating the other possible moves from A.

To evaluate the node C, its left-most child D has to be evaluated first. Let us assume that the value of D is 10 (this value has been obtained either by applying a static evaluation function directly to D, or by backing up values from descendants omitted in the figure). Because this value is less than the value of α , the best move for Max is to node B, independent of the value of node E that need not be evaluated. Indeed, if the value of E is greater than 10, Min will move to D which has the value 10 for Max. Otherwise, if the value of E is less than 10, Min will move to E which has a value less than 10. So, if Max moves to C, the best it can get is 10, which is less than the value $\alpha = 15$ that would be gotten if Max would move to B. Therefore, the best move for Max is to B, independent of the value of E. The elimination of the node E is an alpha cutoff.

One should notice that E may itself have a huge subtree. Therefore, the elimination of E means, in fact, the elimination of this subtree.

A beta cutoff:

To apply this technique, one uses a parameter called beta that represents an upper bound for the achievement of the Max player at a given node.

In the above tree, the Max player moved to the node B. Now it is the turn of the Min player to decide where to move:

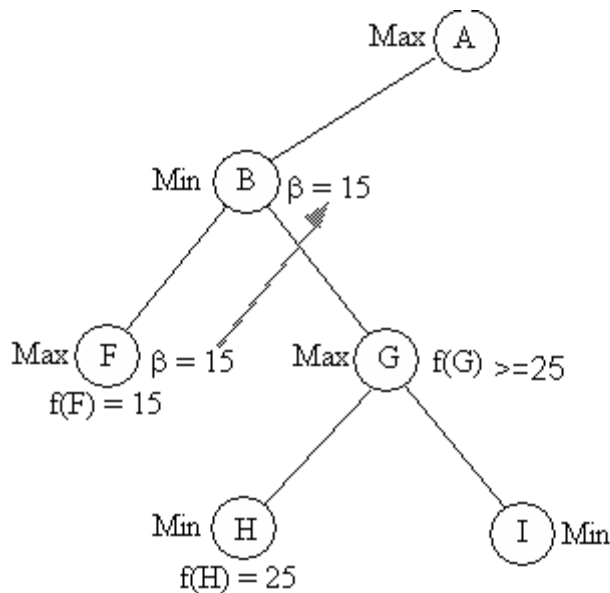


Figure 3.18: Illustration of the beta cut-off.

The Min player also evaluates its descendants in a depth-first order.

Let us assume that the value of F has been evaluated to 15. From the point of view of Min, this is an upper bound for the achievement of Min (it may still be possible to make Min achieve less, depending of the values of the other descendants of B). Therefore the value of β at the node F is 15. This value is transmitted upward to the node B and will be used for evaluating the other possible moves from B.

To evaluate the node G, its left-most child H is evaluated first. Let us assume that the value of H is 25 (this value has been obtained either by applying a static evaluation function directly to H, or by backing up values from descendants omitted in the figure). Because this value is greater than the value of β , the best move for Min is to node F, independent of the value of node I that need not be evaluated. Indeed, if the value of I is $v \geq 25$, then Max (in G) will move to I. Otherwise, if the value of I is less than 25, Max will move to H. So in both cases, the value obtained by Max is at least 25 which is greater than β (the best value obtained by Max if Min moves to F).

Therefore, the best move for Min is at F, independent of the value of I. The elimination of the node I is a beta cutoff.

One should notice that by applying alpha and beta cut-off, one obtains the same results as in the case of mini-max, but (in general) with less effort. This means that, in a given amount of time, one could search deeper in the game tree than in the case of mini-max.

[Unit-5: Knowledge Representation]

Introduction to Artificial Intelligence (CSC-355)

Jagdish Bhatta

**Central Department of Computer Science & Information Technology
Tribhuvan University**

Knowledge Representation

Knowledge:

Knowledge is a theoretical or practical understanding of a subject or a domain. Knowledge is also the sum of what is currently known.

Knowledge is "the sum of what is known: the body of truth, information, and principles acquired by mankind." Or, "Knowledge is what I know, Information is what we know."

There are many other definitions such as:

- Knowledge is "information combined with experience, context, interpretation, and reflection. It is a high-value form of information that is ready to apply to decisions and actions." (T. Davenport et al., 1998)
- Knowledge is "human expertise stored in a person's mind, gained through experience, and interaction with the person's environment." (Sunasee and Sewery, 2002)
- Knowledge is "information evaluated and organized by the human mind so that it can be used purposefully, e.g., conclusions or explanations." (Rousa, 2002)

Knowledge consists of information that has been:

- interpreted,
- categorised,
- applied, experienced and revised.

In general, knowledge is more than just data, it consist of: facts, ideas, beliefs, heuristics, associations, rules, abstractions, relationships, customs.

Research literature classifies knowledge as follows:

Classification-based Knowledge	»	Ability to classify information
Decision-oriented Knowledge	»	Choosing the best option
Descriptive knowledge	»	State of some world (heuristic)
Procedural knowledge	»	How to do something
Reasoning knowledge	»	What conclusion is valid in what situation?
Assimilative knowledge	»	What its impact is?

Knowledge Representation

Knowledge representation (KR) is the study of how knowledge about the world can be represented and what kinds of reasoning can be done with that knowledge. Knowledge Representation is the method used to encode knowledge in Intelligent Systems.

Since knowledge is used to achieve intelligent behavior, the fundamental goal of knowledge representation is to represent knowledge in a manner as to facilitate inferencing (i.e. drawing conclusions) from knowledge. A successful representation of some knowledge must, then, be in a form that is *understandable* by humans, and must cause the system using the knowledge to *behave* as if it knows it.

Some issues that arise in knowledge representation from an AI perspective are:

- How do people represent knowledge?
- What is the nature of knowledge and how do we represent it?
- Should a representation scheme deal with a particular domain or should it be general purpose?
- How expressive is a representation scheme or formal language?
- Should the scheme be declarative or procedural?

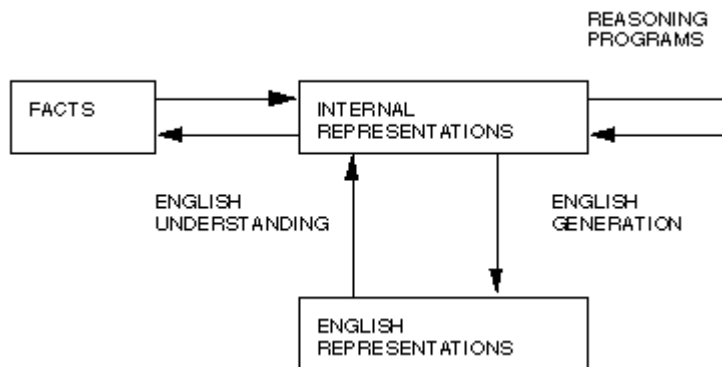


Fig: Two entities in Knowledge Representaion

For example: English or natural language is an obvious way of representing and handling facts. Logic enables us to consider the following fact: *spot is a dog* as $dog(spot)$. We could then infer that all dogs have tails with: $\forall x: dog(x) \rightarrow hasatail(x)$. We can then deduce:

$hasatail(Spot)$

Using an appropriate backward mapping function the English sentence *Spot has a tail can be generated*.

Properties for Knowledge Representation Systems

The following properties should be possessed by a knowledge representation system.

Representational Adequacy

- the ability to represent the required knowledge;

Inferential Adequacy

- the ability to manipulate the knowledge represented to produce new knowledge corresponding to that inferred from the original;

Inferential Efficiency

- the ability to direct the inferential mechanisms into the most productive directions by storing appropriate guides;

Acquisitional Efficiency

- the ability to acquire new knowledge using automatic methods wherever possible rather than reliance on human intervention.

Formal logic-connectives:

In logic, a **logical connective** (also called a **logical operator**) is a symbol or word used to connect two or more sentences (of either a formal or a natural language) in a grammatically valid way, such that the compound sentence produced has a truth value dependent on the respective truth values of the original sentences.

Each logical connective can be expressed as a function, called a truth function. For this reason, logical connectives are sometimes called **truth-functional connectives**.

Commonly used logical connectives include:

- Negation (not) (\neg or \sim)
- Conjunction (and) (\wedge , $\&$, or \cdot)
- Disjunction (or) (\vee or \vee)
- Material implication (if...then) (\rightarrow , \Rightarrow or \supset)
- Biconditional (if and only if) (iff) (xnor) (\leftrightarrow , \equiv , or $=$)

For example, the meaning of the statements *it is raining* and *I am indoors* is transformed when the two are combined with logical connectives:

- It is raining **and** I am indoors ($P \wedge Q$)
- **If** it is raining, **then** I am indoors ($P \rightarrow Q$)
- It is raining **if** I am indoors ($Q \rightarrow P$)
- It is raining **if and only if** I am indoors ($P \leftrightarrow Q$)
- It is **not** raining ($\neg P$)

For statement $P = \text{It is raining}$ and $Q = \text{I am indoors}$.

Truth Table:

A proposition in general contains a number of variables. For example $(P \vee Q)$ contains variables P and Q each of which represents an arbitrary proposition. Thus a proposition takes different values depending on the values of the constituent variables. This relationship of the value of a proposition and those of its constituent variables can be represented by a table. It tabulates the value of a proposition for all possible values of its variables and it is called a truth table.

For example the following table shows the relationship between the values of P , Q and $P \vee Q$:

OR		
P	Q	$(P \vee Q)$
F	F	F
F	T	T
T	F	T
T	T	T

Logic:

Logic is a formal language for representing knowledge such that conclusions can be drawn. **Logic** makes statements about the world which are true (or false) if the state of affairs it represents is the case (or not the case). Compared to natural languages (expressive but context sensitive) and programming languages (good for concrete data structures but not expressive) logic combines the advantages of natural languages and formal languages. Logic is concise, unambiguous, expressive, context insensitive, effective for inferences.

It has syntax, semantics, and proof theory.

Syntax: Describe possible configurations that constitute sentences.

Semantics: Determines what fact in the world, the sentence refers to i.e. the interpretation. Each sentence make claim about the world (meaning of sentence). Semantic property include truth and falsity.

Syntax is concerned with the rules used for constructing, or transforming the symbols and words of a language, as contrasted with the semantics of a language which is concerned with its meaning.

Proof theory (Inference method): set of rules for generating new sentences that are necessarily true given that the old sentences are true.

Artificial Intelligence

We will consider two kinds of logic: **propositional logic** and **first-order logic** or more precisely first-order **predicate calculus**. Propositional logic is of limited expressiveness but is useful to introduce many of the concepts of logic's syntax, semantics and inference procedures.

Entailment:

Entailment means that one thing follows from another:

$$KB \models \alpha$$

Knowledge base KB entails sentence α if and only if α is true in all worlds where KB is true

E.g., $x + y = 4$ entails $4 = x + y$

Entailment is a relationship between sentences (i.e., syntax) that is based on semantics.

We can determine whether $S \models P$ by finding Truth Table for S and P, if any row of Truth Table where all formulae in S is true.

Example:

P	$P \rightarrow Q$	Q
True	True	True
True	False	False
False	True	True
False	True	False

Therefore $\{P, P \rightarrow Q\} \models Q$. Here, only row where both P and $P \rightarrow Q$ are True, Q is also True. Here, $S = \{P, P \rightarrow Q\}$ and $P = \{Q\}$.

Models

Logicians typically think in terms of models, in place of “possible world”, which are formally structured worlds with respect to which truth can be evaluated.

m is a model of a sentence α if α is true in m.

$M(\alpha)$ is the set of all models of α .

Tautology:

A formula of propositional logic is a **tautology** if the formula itself is always true regardless of which valuation is used for the propositional variables.

There are infinitely many tautologies. Examples include:

- $(A \vee \neg A)$ (“A or not A”), the law of the excluded middle. This formula has only one propositional variable, A. Any valuation for this formula must, by definition, assign A one of the truth values *true* or *false*, and assign $\neg A$ the other truth value.

- $(A \rightarrow B) \Leftrightarrow (\neg B \rightarrow \neg A)$ ("if A implies B then not- B implies not- A ", and vice versa), which expresses the law of contraposition.
- $((A \rightarrow B) \wedge (B \rightarrow C)) \rightarrow (A \rightarrow C)$ ("if A implies B and B implies C , then A implies C "), which is the principle known as syllogism.

The definition of **tautology** can be extended to sentences in predicate logic, which may contain quantifiers, unlike sentences of propositional logic. In propositional logic, there is no distinction between a tautology and a **logically valid formula**. In the context of predicate logic, many authors define a tautology to be a sentence that can be obtained by taking a tautology of propositional logic and uniformly replacing each propositional variable by a first-order formula (one formula per propositional variable). The set of such formulas is a proper subset of the set of logically valid sentences of predicate logic (which are the sentences that are true in every model).

There are also propositions that are always false such as $(P \wedge \neg P)$. Such a proposition is called a **contradiction**.

A proposition that is neither a tautology nor a contradiction is called a **contingency**. For example $(P \vee Q)$ is a contingency.

Validity:

The term **validity** in logic (also **logical validity**) is largely synonymous with logical truth, however the term is used in different contexts. Validity is a property of formulae, statements and arguments. **A logically valid argument is one where the conclusion follows from the premises. An invalid argument is where the conclusion does not follow from the premises.** A formula of a formal language is a valid formula if and only if it is true under every possible interpretation of the language.

Saying that an argument is valid is equivalent to saying that it is logically impossible that the premises of the argument are true and the conclusion false. A less precise but intuitively clear way of putting this is to say that in a valid argument IF the premises are true, then the conclusion must be true.

An argument that is not valid is said to be "invalid".

An example of a valid argument is given by the following well-known syllogism:

All men are mortal.

Socrates is a man.

Therefore, Socrates is mortal.

What makes this a valid argument is not that it has true premises and a true conclusion, but the logical necessity of the conclusion, given the two premises.

Artificial Intelligence

The following argument is of the same logical form but with false premises and a false conclusion, and it is equally valid:

All women are cats.
All cats are men.
Therefore, all women are men.

This argument has false premises and a false conclusion. This brings out the hypothetical character of validity. What the validity of these arguments amounts to, is that it assures us the conclusion must be true IF the premises are true.

Thus, an argument is valid if the premises and conclusion follow a logical form. This essentially means that the conclusion logically follows from the premises. An argument is valid if and only if the truth of its premises entails the truth of its conclusion. It would be self-contradictory to affirm the premises and deny the conclusion

Deductive Reasoning:

Deductive reasoning, also called **Deductive logic**, is reasoning which constructs or evaluates deductive arguments. Deductive arguments are attempts to show that a conclusion necessarily follows from a set of premises. **A deductive argument is valid if the conclusion does follow necessarily from the premises, i.e., if the conclusion must be true provided that the premises are true.** A deductive argument is sound if it is valid AND its premises are true. Deductive arguments are valid or invalid, sound or unsound, but are never false or true.

An example of a deductive argument:

1. All men are mortal
2. Socrates is a man
3. Therefore, Socrates is mortal

The first premise states that all objects classified as 'men' have the attribute 'mortal'. The second premise states that 'Socrates' is classified as a man- a member of the set 'men'. The conclusion states that 'Socrates' must be mortal because he inherits this attribute from his classification as a man.

Deductive arguments are generally evaluated in terms of their *validity* and *soundness*. An argument is *valid* if it is impossible both for its premises to be true and its conclusion to be false. An argument can be valid even though the premises are false.

This is an example of a valid argument. The first premise is false, yet the conclusion is still valid.

All fire-breathing rabbits live on Mars

Artificial Intelligence

All humans are fire-breathing rabbits

Therefore, all humans live on Mars

This argument is valid but not *sound*. In order for a deductive argument to be sound, the deduction must be valid and the premise must **all** be true.

Let's take one of the above examples.

1. All monkeys are primates
2. All primates are mammals
3. All monkeys are mammals

This is a sound argument because it is actually true in the real world. The premises are true and so is the conclusion. They logically follow from one another to form a concrete argument that can't be denied. Where validity doesn't have to do with the actual truthfulness of an argument, soundness does.

A theory of deductive reasoning known as categorical or term logic was developed by Aristotle, but was superseded by propositional (sentential) logic and predicate logic.

Deductive reasoning can be contrasted with inductive reasoning. In cases of inductive reasoning, it is possible for the conclusion to be false even though the premises are true and the argument's form is cogent.

Well Formed Formula: (wff)

It is a syntactic object that can be given a semantic meaning. A formal language can be considered to be identical to the set containing all and only its wffs.

A key use of wffs is in propositional logic and predicate logics such as first-order logic. In those contexts, a formula is a string of symbols ϕ for which it makes sense to ask "is ϕ true?", once any free variables in ϕ have been instantiated. In formal logic, proofs can be represented by sequences of wffs with certain properties, and the final wff in the sequence is what is proven.

The well-formed formulas of **propositional calculus** are expressions such as $(A \wedge (B \vee C))$. Their definition begins with the arbitrary choice of a set V of propositional variables. The alphabet consists of the letters in V along with the symbols for the propositional connectives and parentheses "(" and ")", all of which are assumed to not be in V . The wffs will be certain expressions (that is, strings of symbols) over this alphabet.

The well-formed formulas are inductively defined as follows:

- Each propositional variable is, on its own, a wff.
- If ϕ is a wff, then $\neg\phi$ is a wff.

- If ϕ and ψ are wffs, and \bullet is any binary connective, then $(\phi \bullet \psi)$ is a wff. Here \bullet could be $\vee, \wedge, \rightarrow$, or \leftrightarrow .

The WFF for **predicate calculus** is defined to be the smallest set containing the set of atomic WFFs such that the following holds:

1. $\neg\phi$ is a WFF when ϕ is a WFF
2. $(\phi \wedge \psi)$ and $(\phi \vee \psi)$ are WFFs when ϕ and ψ are WFFs;
3. $\exists x \phi$ is a WFF when x is a variable and ϕ is a WFF;
4. $\forall x \phi$ is a WFF when x is a variable and ϕ is a WFF (alternatively, $\forall x \phi$ could be defined as an abbreviation for $\neg \exists x \neg \phi$).

If a formula has no occurrences of $\exists x$ or $\forall x$, for any variable x , then it is called *quantifier-free*. An *existential formula* is a string of existential quantification followed by a quantifier-free formula.

Propositional Logic:

Propositional logic represents knowledge/ information in terms of propositions. Propositions are facts and non-facts that can be true or false. Propositions are expressed using ordinary declarative sentences. Propositional logic is the simplest logic.

Syntax:

The syntax of propositional logic defines the allowable sentences. The atomic sentences- the indivisible syntactic elements- consist of single proposition symbol. Each such symbol stands for a proposition that can be true or false. We use the symbols like P1, P2 to represent sentences.

The complex sentences are constructed from simpler sentences using logical connectives. There are five connectives in common use:

\neg (*negation*), \wedge (*conjunction*), \vee (*disjunction*), \Rightarrow (*implication*), \Leftrightarrow (*biconditional*)

The order of precedence in propositional logic is from (highest to lowest): $\neg, \wedge, \vee, \Rightarrow, \Leftrightarrow$.

Propositional logic is defined as:

If S is a sentence, $\neg S$ is a sentence (*negation*)

If S1 and S2 are sentences, $S1 \wedge S2$ is a sentence (*conjunction*)

If S1 and S2 are sentences, $S1 \vee S2$ is a sentence (*disjunction*)

If S1 and S2 are sentences, $S1 \Rightarrow S2$ is a sentence (*implication*)

If S1 and S2 are sentences, $S1 \Leftrightarrow S2$ is a sentence (*biconditional*)

Artificial Intelligence

Formal grammar for propositional logic can be given as below:

Sentence \rightarrow AtomicSentence | ComplexSentence

AtomicSentence \rightarrow True | False | Symbol

Symbol \rightarrow P | Q | R

ComplexSentence \rightarrow \neg Sentence

$|$ (Sentence \wedge Sentence)

$|$ (Sentence \vee Sentence)

$|$ (Sentence \Rightarrow Sentence)

$|$ (Sentence \Leftrightarrow Sentence)

Semantics:

Each model specifies true/false for each proposition symbol

Rules for evaluating truth with respect to a model:

\neg S is true if, S is false

$S1 \wedge S2$ is true if, S1 is true and S2 is true

$S1 \vee S2$ is true if, S1 is true or S2 is true

$S1 \Rightarrow S2$ is true if, S1 is false or S2 is true

$S1 \Leftrightarrow S2$ is true if, $S1 \Rightarrow S2$ is true and $S2 \Rightarrow S1$ is true

Truth Table showing the evaluation of semantics of complex sentences:

P	Q	\neg P	$P \wedge Q$	$P \vee Q$	$P \Rightarrow Q$	$P \Leftrightarrow Q$
false	false	true	false	false	true	true
false	true	true	false	true	true	false
true	false	false	false	true	false	false
true	true	false	true	true	true	true

Logical equivalence:

Artificial Intelligence

Two sentences α and β are *logically equivalent* ($\alpha \equiv \beta$) iff true they are true in same set of models
or Two sentences α and β are *logically equivalent* ($\alpha \equiv \beta$) iff $\alpha \models \beta$ and $\beta \models \alpha$.

$$\begin{aligned}(\alpha \wedge \beta) &\equiv (\beta \wedge \alpha) && \text{commutativity of } \wedge \\(\alpha \vee \beta) &\equiv (\beta \vee \alpha) && \text{commutativity of } \vee \\((\alpha \wedge \beta) \wedge \gamma) &\equiv (\alpha \wedge (\beta \wedge \gamma)) && \text{associativity of } \wedge \\((\alpha \vee \beta) \vee \gamma) &\equiv (\alpha \vee (\beta \vee \gamma)) && \text{associativity of } \vee \\\neg(\neg\alpha) &\equiv \alpha && \text{double-negation elimination} \\(\alpha \Rightarrow \beta) &\equiv (\neg\beta \Rightarrow \neg\alpha) && \text{contraposition} \\(\alpha \Rightarrow \beta) &\equiv (\neg\alpha \vee \beta) && \text{implication elimination} \\(\alpha \Leftrightarrow \beta) &\equiv ((\alpha \Rightarrow \beta) \wedge (\beta \Rightarrow \alpha)) && \text{biconditional elimination} \\\neg(\alpha \wedge \beta) &\equiv (\neg\alpha \vee \neg\beta) && \text{de Morgan} \\\neg(\alpha \vee \beta) &\equiv (\neg\alpha \wedge \neg\beta) && \text{de Morgan} \\(\alpha \wedge (\beta \vee \gamma)) &\equiv ((\alpha \wedge \beta) \vee (\alpha \wedge \gamma)) && \text{distributivity of } \wedge \text{ over } \vee \\(\alpha \vee (\beta \wedge \gamma)) &\equiv ((\alpha \vee \beta) \wedge (\alpha \vee \gamma)) && \text{distributivity of } \vee \text{ over } \wedge\end{aligned}$$

Validity:

A sentence is *valid* if it is true in all models,

$$\text{e.g., True, } A \vee \neg A, A \Rightarrow A, (A \wedge (A \Rightarrow B)) \Rightarrow B$$

Valid sentences are also known as tautologies. Every valid sentence is logically equivalent to True

Satisfiability:

A sentence is *satisfiable* if it is true in *some* model

$$\text{— e.g., } A \vee B, C$$

A sentence is *unsatisfiable* if it is true in *no* models

$$\text{— e.g., } A \wedge \neg A$$

Validity and satisfiability are related concepts

- α is valid iff $\neg\alpha$ is unsatisfiable
- α is satisfiable iff $\neg\alpha$ is not valid

Satisfiability is connected to inference via the following:

- $KB \models \alpha$ if and only if $(KB \wedge \neg\alpha)$ is unsatisfiable

Inference rules in Propositional Logic

Inference rules are the standard patterns of inference that can be applied to derive conclusions from given facts.

Modus Ponens

$$\frac{\alpha \Rightarrow \beta, \alpha}{\beta}$$

And-elimination

$$\frac{\alpha \wedge \beta}{\alpha}$$

Monotonicity: the set of entailed sentences can only increase as information is added to the knowledge base.

For any sentence α and β if $KB \models \alpha$ then $KB \wedge \beta \models \alpha$.

Resolution

Unit resolution rule:

Unit resolution rule takes a clause – a disjunction of literals – and a literal and produces a new clause. Single literal is also called unit clause.

$$\frac{\ell_1 \vee \dots \vee \ell_k, \quad m}{\ell_1 \vee \dots \vee \ell_{i-1} \vee \ell_{i+1} \vee \dots \vee \ell_k}$$

Where ℓ_i and m are complementary literals

Generalized resolution rule:

Generalized resolution rule takes two clauses of any length and produces a new clause as below.

$$\frac{\ell_1 \vee \dots \vee \ell_k, \quad m_1 \vee \dots \vee m_n}{\ell_1 \vee \dots \vee \ell_{i-1} \vee \ell_{i+1} \vee \dots \vee \ell_k \vee m_1 \vee \dots \vee m_{j-1} \vee m_{j+1} \vee \dots \vee m_n}$$

For example:

$$\frac{\ell_1 \vee \ell_2, \quad \neg \ell_2 \vee \ell_3}{\ell_1 \vee \ell_3}$$

Resolution Uses CNF (Conjunctive normal form)

- **Conjunction of disjunctions of literals (clauses)**

The resolution rule is sound:

- Only entailed sentences are derived

Resolution is complete in the sense that it can always be used to either confirm or refute a sentence (it can not be used to enumerate true sentences.)

Conversion to CNF:

A sentence that is expressed as a conjunction of disjunctions of literals is said to be in conjunctive normal form (CNF). A sentence in CNF that contains only k literals per clause is said to be in k -CNF.

Algorithm:

Eliminate \leftrightarrow rewriting $P \leftrightarrow Q$ as $(P \rightarrow Q) \wedge (Q \rightarrow P)$

Eliminate \rightarrow rewriting $P \rightarrow Q$ as $\neg P \vee Q$

Use De Morgan's laws to push \neg inwards:

- rewrite $\neg (P \vee Q)$ as $\neg P \wedge \neg Q$

- rewrite $\neg (P \wedge Q)$ as $\neg P \vee \neg Q$

Eliminate double negations: rewrite $\neg \neg P$ as P

Use the distributive laws to get CNF:

- rewrite $(P \wedge Q) \vee R$ as $(P \vee R) \wedge (Q \vee R)$

Flatten nested clauses:

- $(P \vee Q) \vee R$ as $P \vee Q \vee R$

- $(P \vee Q) \vee R$ as $P \vee Q \vee R$

Example: Let's illustrate the conversion to CNF by using an example.

$$B \Leftrightarrow (A \vee C)$$

- Eliminate \Leftrightarrow , replacing $\alpha \Leftrightarrow \beta$ with $(\alpha \Rightarrow \beta) \wedge (\beta \Rightarrow \alpha)$.
 - $(B \Rightarrow (A \vee C)) \wedge ((A \vee C) \Rightarrow B)$
- Eliminate \Rightarrow , replacing $\alpha \Rightarrow \beta$ with $\neg \alpha \vee \beta$.
 - $(\neg B \vee A \vee C) \wedge (\neg(A \vee C) \vee B)$
- Move \neg inwards using de Morgan's rules and double-negation:
 - $(\neg B \vee A \vee C) \wedge ((\neg A \wedge \neg C) \vee B)$
- Apply distributivity law (\wedge over \vee) and flatten:
 - $(\neg B \vee A \vee C) \wedge (\neg A \vee B) \wedge (\neg C \vee B)$

Resolution algorithm

- Convert KB into CNF
- Add negation of sentence to be entailed into KB i.e. $(KB \wedge \neg \alpha)$
- Then apply resolution rule to resulting clauses.

Artificial Intelligence

- The process continues until:
 - There are no new clauses that can be added
Hence KB **does not** entail α
 - Two clauses resolve to entail the empty clause.
Hence KB **does** entail α

Example: Consider the knowledge base given as: $KB = (B \Leftrightarrow (A \vee C)) \wedge \neg B$

Prove that $\neg A$ can be inferred from above KB by using resolution.

Solution:

At first, convert KB into CNF

$$B \Rightarrow (A \vee C) \wedge ((A \vee C) \Rightarrow B) \wedge \neg B$$

$$(\neg B \vee A \vee C) \wedge (\neg(A \vee C) \vee B) \wedge \neg B$$

$$(\neg B \vee A \vee C) \wedge ((\neg A \wedge \neg C) \vee B) \wedge \neg B$$

$$(\neg B \vee A \vee C) \wedge (\neg A \vee B) \wedge (\neg C \vee B) \wedge \neg B$$

Add negation of sentence to be inferred from KB into KB

Now KB contains following sentences all in CNF

$$(\neg B \vee A \vee C)$$

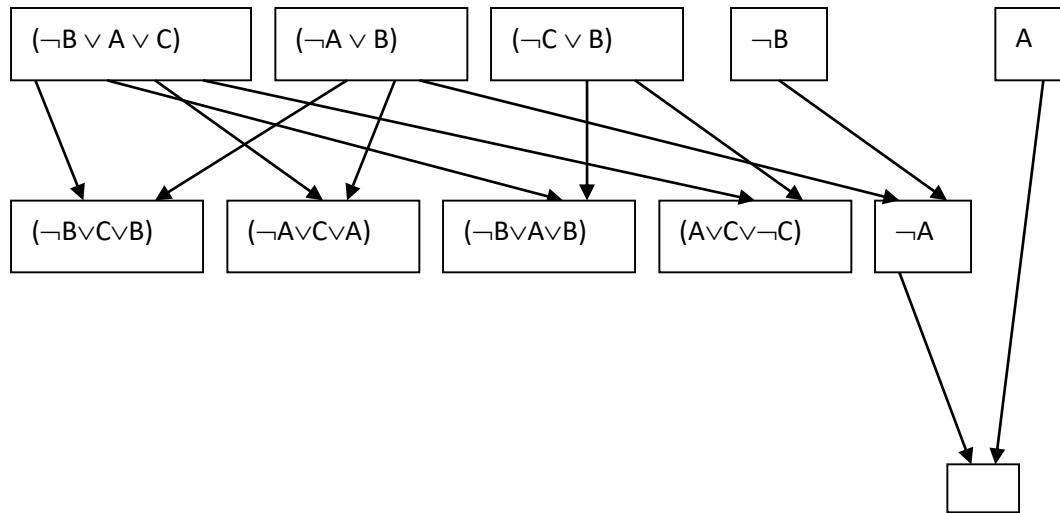
$$(\neg A \vee B)$$

$$(\neg C \vee B)$$

$$\neg B$$

A (negation of conclusion to be proved)

Now use Resolution algorithm



Resolution: More Examples

1. $KB = \{(GVH) \rightarrow (\neg J \wedge \neg K), G\}$. Show that $KB \vdash \neg J$

Solution:

Clausal form of $(GVH) \rightarrow (\neg J \wedge \neg K)$ is

$\{\neg G \vee \neg J, \neg H \vee \neg J, \neg G \vee \neg K, \neg H \vee \neg K\}$

1. $\neg G \vee \neg J$ [Premise]

2. $\neg H \vee \neg J$ [Premise]

3. $\neg G \vee \neg K$ [Premise]

4. $\neg H \vee \neg K$ [Premise]

5. G [Premise]

6. J [\neg Conclusion]

7. $\neg G$ [1, 6 Resolution]

8. $_$ [5, 7 Resolution]

Hence KB entails $\neg J$

2. $KB = \{P \rightarrow \neg Q, \neg Q \rightarrow R\}$. Show that $KB \vdash P \rightarrow R$

Solution:

1. $\neg P \vee \neg Q$ [Premise]
2. $Q \vee R$ [Premise]
3. P [\neg Conclusion]
4. $\neg R$ [\neg Conclusion]
5. $\neg Q$ [1, 3 Resolution]
6. R [2, 5 Resolution]
7. $_$ [4, 6 Resolution]

Hence, $KB \vdash P \rightarrow R$

3. $\vdash ((P \vee Q) \wedge \neg P) \rightarrow Q$

Clausal form of $\neg (((P \vee Q) \wedge \neg P) \rightarrow Q)$ is $\{P \vee Q, \neg P, \neg Q\}$

1. $P \vee Q$ [\neg Conclusion]
2. $\neg P$ [\neg Conclusion]
3. $\neg Q$ [\neg Conclusion]
4. Q [1, 2 Resolution]
5. $_$ [3, 4 Resolution]

Forward and backward chaining

The completeness of resolution makes it a very important inference model. But in many practical situations full power of resolution is not needed. Real-world knowledge bases often contain only clauses of restricted kind called **Horn Clause**. A Horn clause is disjunction of literals with at most one positive literal

Three important properties of Horn clause are:

- ✓ Can be written as an implication
- ✓ Inference through forward chaining and backward chaining.
- ✓ Deciding entailment can be done in a time linear size of the knowledge base.

Forward chaining:

Idea: fire any rule whose premises are satisfied in the *KB*,

- add its conclusion to the *KB*, until query is found

$$P \Rightarrow Q$$

$$L \wedge M \Rightarrow P$$

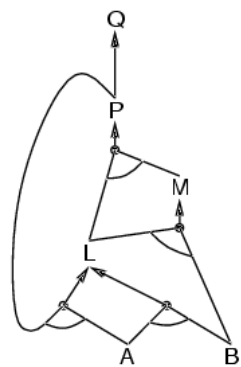
$$B \wedge L \Rightarrow M$$

$$A \wedge P \Rightarrow L$$

$$A \wedge B \Rightarrow L$$

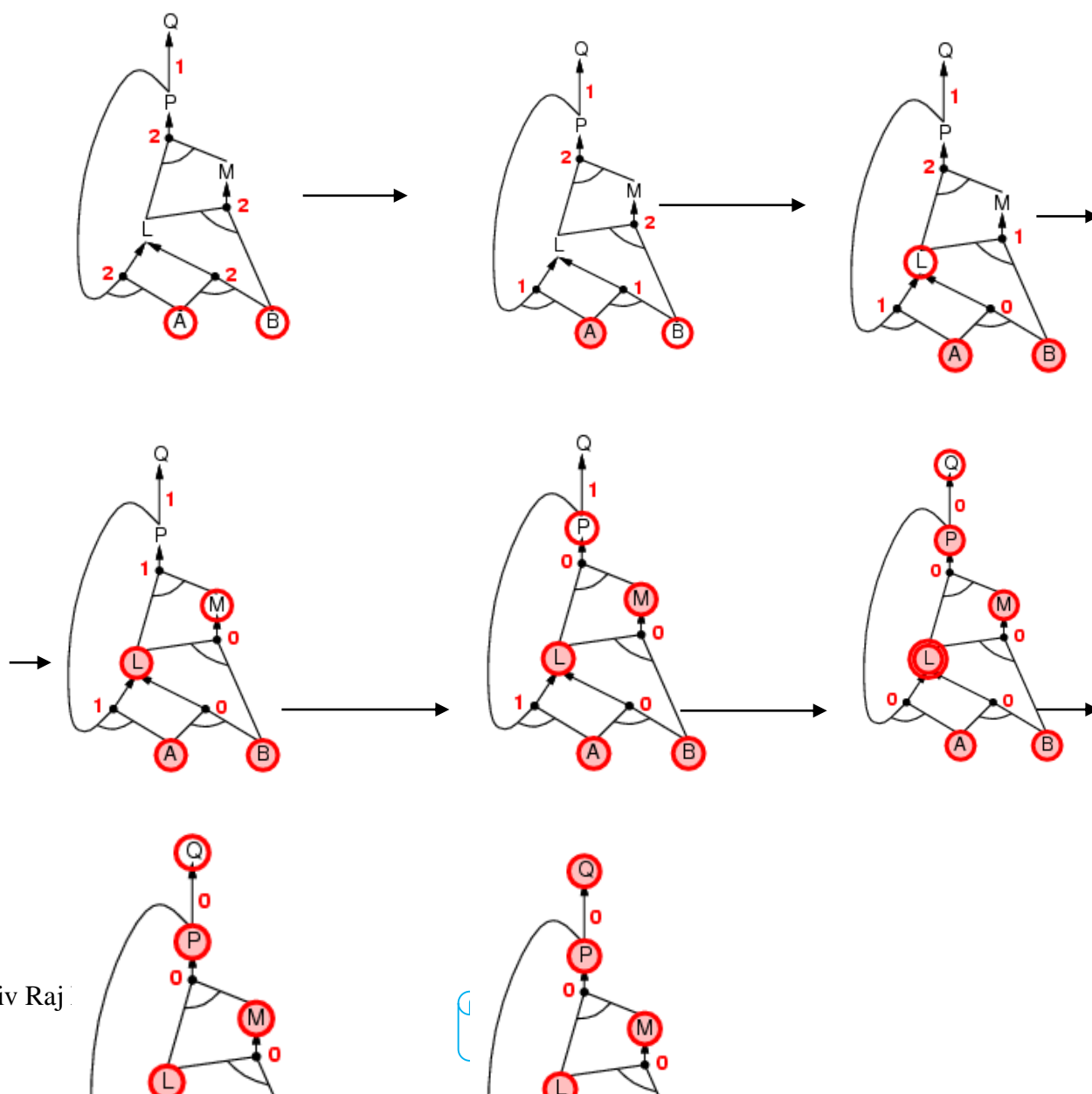
A

B



Prove that *Q* can be inferred from above *KB*

Solution:



Artificial Intelligence

Backward chaining:

Idea: work backwards from the query q : to prove q by BC,

Check if q is known already, or

Prove by BC all premises of some rule concluding q

For example, for above KB (as in forward chaining above)

$$P \Rightarrow Q$$

$$L \wedge M \Rightarrow P$$

$$B \wedge L \Rightarrow M$$

$$A \wedge P \Rightarrow L$$

$$A \wedge B \Rightarrow L$$

A

B

Prove that Q can be inferred from above KB

Solution:

We know $P \Rightarrow Q$, try to prove P

$$L \wedge M \Rightarrow P$$

Try to prove L and M

$$B \wedge L \Rightarrow M$$

$$A \wedge P \Rightarrow L$$

Try to prove B , L and A and P

A and B is already known, since $A \wedge B \Rightarrow L$, L is also known

Since, $B \wedge L \Rightarrow M$, M is also known

Since, $L \wedge M \Rightarrow P$, p is known, hence the **proved**.

First-Order Logic

Pros and cons of propositional logic

Artificial Intelligence

- Propositional logic is declarative
- Propositional logic allows partial/disjunctive/negated information
 - o (unlike most data structures and databases)
- Propositional logic is compositional:
 - o meaning of $B \wedge P$ is derived from meaning of B and of P
- Meaning in propositional logic is context-independent
 - o (unlike natural language, where meaning depends on context)
- Propositional logic has very limited expressive power
 - o (unlike natural language)

Propositional logic assumes the world contains facts, whereas first-order logic (like natural language) assumes the world contains:

- Objects: people, houses, numbers, colors, baseball games, wars, ...
- Relations: red, round, prime, brother of, bigger than, part of, comes between,...
- Functions: father of, best friend, one more than, plus, ...

Logics in General

The primary difference between PL and FOPL is their ontological commitment:

Ontological Commitment: What exists in the world — TRUTH

- PL: facts hold or do not hold.
- FL : objects with relations between them that hold or do not hold

Another difference is:

Epistemological Commitment: What an agent believes about facts — BELIEF

Language	Ontological Commitment	Epistemological Commitment
Propositional logic	facts	true/false/unknown
First-order logic	facts, objects, relations	true/false/unknown
Temporal logic	facts, objects, relations, times	true/false/unknown
Probability theory	facts	degree of belief $\in [0, 1]$
Fuzzy logic	degree of truth $\in [0, 1]$	known interval value

FOPL: Syntax

Predicate Logic: Syntax

<i>Sentence</i>	→	<i>AtomicSentence</i>
		<i>(Sentence</i> <i>Connective</i> <i>Sentence)</i>
		<i>Quantifier Variable, ... Sentence</i>
		<i>¬ Sentence</i>
<i>AtomicSentence</i>	→	<i>Predicate(Term, ...)</i> <i>Term = Term</i>
<i>Term</i>	→	<i>Function(Term, ...)</i> <i>Constant</i> <i>Variable</i>
<i>Connective</i>	→	\wedge \vee \Rightarrow \Leftrightarrow
<i>Quantifier</i>	→	\forall \exists
<i>Constant</i>	→	<i>A, B, C, X₁, X₂, Jim, Jack</i>
<i>Variable</i>	→	<i>a, b, c, x₁, x₂, counter, position, ...</i>
<i>Predicate</i>	→	<i>Adjacent-To, Younger-Than, HasColor, ...</i>
<i>Function</i>	→	<i>Father-Of, Square-Position, Sqrt, Cosine</i>

ambiguities are resolved through precedence or parentheses

Representing knowledge in first-order logic

The objects from the real world are represented by constant symbols (a,b,c,...). For instance, the symbol "Tom" may represent a certain individual called Tom.

Properties of objects may be represented by predicates applied to those objects ($P(a)$, ...): e.g "male(Tom)" represents that Tom is a male.

Relationships between objects are represented by predicates with more arguments: "father(Tom, Bob)" represents the fact that Tom is the father of Bob.

The value of a predicate is one of the boolean constants T (i.e. true) or F (i.e. false). "father(Tom, Bob) = T" means that the sentence "Tom is the father of Bob" is true. "father(Tom, Bob) = F" means that the sentence "Tom is the father of Bob" is false.

Besides constants, the arguments of the predicates may be functions (f,g,...) or variables (x,y,...).

Function symbols denote mappings from elements of a domain (or tuples of elements of domains) to elements of a domain. For instance, weight is a function that maps objects to their weight: weight (Tom) = 150. Therefore the predicate greater-than (weight (Bob), 100) means that the weight of Bob is greater than 100. The arguments of a function may themselves be functions.

Artificial Intelligence

Variable symbols represent potentially any element of a domain and allow the formulation of general statements about the elements of the domain.

The quantifier's \forall and \exists are used to build new formulas from old ones.

" $\exists x P(x)$ " expresses that there is at least one element of the domain that makes $P(x)$ true.

" $\exists x \text{mother}(x, \text{Bob})$ " means that there is x such that x is mother of Bob or, otherwise stated, Bob has a mother.

" $\forall x P(x)$ " expresses that for all elements of the domain $P(x)$ is true.

Quantifiers

Allows us to express properties of collections of objects instead of enumerating objects by name. Two quantifiers are:

Universal: "for all" \forall

Existential: "there exists" \exists

Universal quantification:

$\forall \langle \text{Variables} \rangle \langle \text{sentence} \rangle$

Eg: Everyone at UAB is smart:

$$\forall x \text{At}(x, \text{UAB}) \Rightarrow \text{Smart}(x)$$

$\forall x P$ is true in a model m iff P is true for all x in the model

Roughly speaking, equivalent to the conjunction of instantiations of P

$\text{At}(\text{KingJohn}, \text{UAB}) \Rightarrow \text{Smart}(\text{KingJohn}) \wedge \text{At}(\text{Richard}, \text{UAB}) \Rightarrow \text{Smart}(\text{Richard}) \wedge \text{At}(\text{UAB}, \text{UAB}) \Rightarrow \text{Smart}(\text{UAB}) \wedge \dots$

Typically, \Rightarrow is the main connective with \forall

– A universally quantifier is also equivalent to a set of implications over all objects
Common mistake: using \wedge as the main connective with \forall :

$$\forall x \text{At}(x, \text{UAB}) \wedge \text{Smart}(x)$$

Means "Everyone is at UAB and everyone is smart"

Existential quantification

$\exists \langle \text{variables} \rangle \langle \text{sentence} \rangle$

Someone at UAB is smart:

Artificial Intelligence

$$\exists x \text{ At}(x, \text{UAB}) \wedge \text{Smart}(x)$$

$\exists x$ P is true in a model m iff P is true for at least one x in the model

Roughly speaking, equivalent to the disjunction of instantiations of P

$$\text{At}(\text{KingJohn}, \text{UAB}) \wedge \text{Smart}(\text{KingJohn}) \vee \text{At}(\text{Richard}, \text{UAB}) \wedge \text{Smart}(\text{Richard})$$

$$\vee \text{At}(\text{UAB}, \text{UAB}) \wedge \text{Smart}(\text{UAB}) \vee \dots$$

Typically, \wedge is the main connective with \exists

Common mistake: using \Rightarrow as the main connective with \exists :

$$\exists x \text{ At}(x, \text{UAB}) \Rightarrow \text{Smart}(x) \text{ is true even if there is anyone who is not at UAB!}$$

FOPL: Semantic

An interpretation is required to give semantics to first-order logic. The interpretation is a non-empty “domain of discourse” (set of objects). The truth of any formula depends on the interpretation.

The interpretation provides, for each:

constant symbol an object in the domain

function symbols a function from domain tuples to the domain

predicate symbol a relation over the domain (a set of tuples)

Then we define:

universal quantifier $\forall x P(x)$ is True iff $P(a)$ is True for all assignments of domain elements a to x

existential quantifier $\exists x P(x)$ is True iff $P(a)$ is True for at least one assignment of domain element a to x

FOPL: Inference (Inference in first-order logic)

First order inference can be done by converting the knowledge base to PL and using propositional inference.

- How to convert universal quantifiers?
 - Replace variable by ground term.
- How to convert existential quantifiers?
 - Skolemization.

Universal instantiation (UI)

Substitute ground term (term without variables) for the variables.

For example consider the following KB

$$\forall x \text{ King}(x) \wedge \text{Greedy}(x) \Rightarrow \text{Evil}(x)$$

King (John)

Greedy (John)

Brother (Richard, John)

It's UI is:

$$\text{King}(\text{John}) \wedge \text{Greedy}(\text{John}) \Rightarrow \text{Evil}(\text{John})$$

$$\text{King}(\text{Richard}) \wedge \text{Greedy}(\text{Richard}) \Rightarrow \text{Evil}(\text{Richard})$$

King (John)

Greedy (John)

Brother (Richard, John)

Note: Remove universally quantified sentences after universal instantiation.

Existential instantiation (EI)

For any sentence α and variable v in that, introduce a constant that is not in the KB (called skolem constant) and substitute that constant for v .

E.g.: Consider the sentence, $\exists x \text{ Crown}(x) \wedge \text{OnHead}(x, \text{John})$

After EI,

$$\text{Crown}(C1) \wedge \text{OnHead}(C1, \text{John}) \quad \text{where } C1 \text{ is Skolem Constant.}$$

Towards Resolution for FOPL:

- Based on resolution for propositional logic
- Extended syntax: allow variables and quantifiers
- Define "clausal form" for first-order logic formulae (CNF)
- Eliminate quantifiers from clausal forms
- Adapt resolution procedure to cope with variables (unification)

Conversion to CNF:

1. Eliminate implications and bi-implications as in propositional case

2. Move negations inward using De Morgan's laws

plus rewriting $\neg \forall x P$ as $\exists x \neg P$ and $\neg \exists x P$ as $\forall x \neg P$

3. Eliminate double negations

4. Rename bound variables if necessary so each only occurs once

e.g. $\forall x P(x) \vee \exists x Q(x)$ becomes $\forall x P(x) \vee \exists y Q(y)$

5. Use equivalences to move quantifiers to the left

e.g. $\forall x P(x) \wedge Q$ becomes $\forall x (P(x) \wedge Q)$ where x is not in Q

e.g. $\forall x P(x) \wedge \exists y Q(y)$ becomes $\forall x \exists y (P(x) \wedge Q(y))$

6. Skolemise (replace each existentially quantified variable by a **new** term)

$\exists x P(x)$ becomes $P(a_0)$ using a Skolem constant a_0 since $\exists x$ occurs at the outermost level

$\forall x \exists y P(x, y)$ becomes $P(x, f_0(x))$ using a Skolem function f_0 since $\exists y$ occurs within $\forall x$

7. The formula now has only universal quantifiers and all are at the left of the formula: drop them

8. Use distribution laws to get CNF and then clausal form

Example:

1.) $\forall x [\forall y P(x, y) \rightarrow \neg \forall y (Q(x, y) \rightarrow R(x, y))]$

Solution:

1. $\forall x [\neg \forall y P(x, y) \vee \neg \forall y (\neg Q(x, y) \vee R(x, y))]$

2, 3. $\forall x [\exists y \neg P(x, y) \vee \exists y (Q(x, y) \wedge \neg R(x, y))]$

4. $\forall x [\exists y \neg P(x, y) \vee \exists z (Q(x, z) \wedge \neg R(x, z))]$

5. $\forall x \exists y \exists z [\neg P(x, y) \vee (Q(x, z) \wedge \neg R(x, z))]$

6. $\forall x [\neg P(x, f(x)) \vee (Q(x, g(x)) \wedge \neg R(x, g(x)))]$

$$7. \neg P(x, f(x)) \vee (Q(x, g(x)) \wedge \neg R(x, g(x)))$$

$$8. (\neg P(x, f(x)) \vee Q(x, g(x))) \wedge (\neg P(x, f(x)) \vee \neg R(x, g(x)))$$

$$8. \{ \neg P(x, f(x)) \vee Q(x, g(x)), \neg P(x, f(x)) \vee \neg R(x, g(x)) \}$$

$$2.) \neg \exists x \forall y \forall z ((P(y) \vee Q(z)) \rightarrow (P(x) \vee Q(x)))$$

Solution:

$$1. \neg \exists x \forall y \forall z (\neg (P(y) \vee Q(z)) \vee P(x) \vee Q(x))$$

$$2. \forall x \neg \forall y \forall z (\neg (P(y) \vee Q(z)) \vee P(x) \vee Q(x))$$

$$2. \forall x \exists y \neg \forall z (\neg (P(y) \vee Q(z)) \vee P(x) \vee Q(x))$$

$$2. \forall x \exists y \exists z \neg (\neg (P(y) \vee Q(z)) \vee P(x) \vee Q(x))$$

$$2. \forall x \exists y \exists z ((P(y) \vee Q(z)) \wedge \neg (P(x) \vee Q(x)))$$

$$6. \forall x ((P(f(x)) \vee Q(g(x))) \wedge \neg P(x) \wedge \neg Q(x))$$

$$7. (P(f(x)) \vee Q(g(x))) \wedge \neg P(x) \wedge \neg Q(x)$$

$$8. \{P(f(x)) \vee Q(g(x)), \neg P(x), \neg Q(x)\}$$

Unification:

A unifier of two atomic formulae is a substitution of terms **for variables** that makes them identical.

- Each variable has at most one associated term

- Substitutions are applied simultaneously

Unifier of $P(x, f(a), z)$ and $P(z, z, u) : \{x/f(a), z/f(a), u/f(a)\}$

We can get the inference immediately if we can find a substitution α such that $King(x)$ and $Greedy(x)$ match $King(John)$ and $Greedy(y)$

$\alpha = \{x/John, y/John\}$ works

$Unify(\alpha, \beta) = \theta$ if $\alpha\theta = \theta\beta$

$$\begin{array}{ccc} p & q & \theta \\ \text{Knows}(\text{John}, x) & \text{Knows}(\text{John}, \text{Jane}) & \{x/\text{Jane}\} \end{array}$$

Artificial Intelligence

Knows(John,x)	Knows(y,OJ)	{x/OJ, y/John}
Knows(John,x)	Knows(y,Mother(y))	{y/John, x/Mother(John)}
Knows(John,x)	Knows(x,OJ)	{fail}

Last unification is failed due to overlap of variables. x can not take the values of John and OJ at the same time.

We can avoid this problem by renaming to avoid the name clashes (standardizing apart)

E.g.

Unify{Knows(John,x) Knows(z,OJ) } = {x/OJ, z/John}

Another complication:

To unify *Knows(John,x)* and *Knows(y,z)*,

Unification of *Knows(John,x)* and *Knows(y,z)* gives $\alpha = \{y/\text{John}, x/z\}$ or $\alpha = \{y/\text{John}, x/\text{John}, z/\text{John}\}$

First unifier gives the result *Knows(John,z)* and second unifier gives the result *Knows(John, John)*. Second can be achieved from first by substituting john in place of z. The first unifier is more general than the second.

There is a single most general unifier (MGU) that is unique up to renaming of variables.

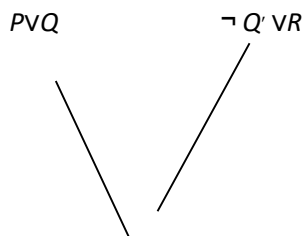
MGU = { y/John, x/z }

Towards Resolution for First-Order Logic

- Based on resolution for propositional logic
- Extended syntax: allow variables and quantifiers
- Define “clausal form” for first-order logic formulae
- Eliminate quantifiers from clausal forms
- Adapt resolution procedure to cope with variables (unification)

First-Order Resolution

For clauses $P \vee Q$ and $\neg Q' \vee R$ with Q, Q' atomic formulae



$$(P \vee R)\theta$$

where θ is a most general unifier for Q and Q'

$(P \vee R)\theta$ is the resolvent of the two clauses

Applying Resolution Refutation

- Negate query to be proven (resolution is a refutation system)
- Convert knowledge base and negated query into CNF and extract clauses
- Repeatedly apply resolution to clauses or copies of clauses until either the empty clause (contradiction) is derived or no more clauses can be derived (a copy of a clause is the clause with all variables renamed)
- If the empty clause is derived, answer 'yes' (query follows from knowledge base), otherwise answer 'no' (query does not follow from knowledge base)

Resolution: Examples

$$1.) \vdash \exists x (P(x) \rightarrow \forall x P(x))$$

Solution:

Add negation of the conclusion and convert the predicate in to CNF:

$$(\neg \exists x (P(x) \rightarrow \forall x P(x)))$$

$$1, 2. \forall x \neg (\neg P(x) \vee \forall x P(x))$$

$$2. \forall x (\neg \neg P(x) \wedge \neg \forall x P(x))$$

$$2, 3. \forall x (P(x) \wedge \exists x \neg P(x))$$

$$4. \forall x (P(x) \wedge \exists y \neg P(y))$$

$$5. \forall x \exists y (P(x) \wedge \neg P(y))$$

$$6. \forall x (P(x) \wedge \neg P(f(x)))$$

$$8. P(x), \neg P(f(x))$$

Artificial Intelligence

Now, we can use resolution as;

1. $P(x)$ [\neg Conclusion]
2. $\neg P(f(y))$ [Copy of \neg Conclusion]
3. $_$ [1, 2 Resolution $\{x/f(y)\}$]

2.) $\vdash \exists x \forall y \forall z ((P(y) \vee Q(z)) \rightarrow (P(x) \vee Q(x)))$

Solution:

1. $P(f(x)) \vee Q(g(x))$ [\neg Conclusion]
2. $\neg P(x)$ [\neg Conclusion]
3. $\neg Q(x)$ [\neg Conclusion]
4. $\neg P(y)$ [Copy of 2]
5. $Q(g(x))$ [1, 4 Resolution $\{y/f(x)\}$]
6. $\neg Q(z)$ [Copy of 3]
7. $_$ [5, 6 Resolution $\{z/g(x)\}$]

The following axioms describe the situation:

1. If the coin comes up heads, then I win.
2. If it comes up tails, then you lose.
3. If it does not come up heads, then it comes up tails.
4. if you lose, then I win.

Which may be represented as:

1. $H \rightarrow W(\text{me})$ //H: heads , W: win
2. $T \rightarrow L(\text{you})$ //T: tails, L: lose
3. $\neg H \rightarrow T$
4. $L(\text{you}) \rightarrow W(\text{me})$

Next, our argument is converted to clause form

1. $\neg H \vee W(\text{me})$
2. $\neg T \vee L(\text{you})$
3. $H \vee T$
4. $\neg L(\text{you}) \vee W(\text{me})$

Then, add the negation of the conclusion

5. $\neg W(\text{me})$ //also in clause form

Finally, we attempt to obtain a contradiction

- | | | |
|-----|----------------------------|-------------------------|
| 2,4 | $\neg T \vee W(\text{me})$ | 6 |
| 1,3 | $T \vee W(\text{me})$ | 7 |
| 6,7 | $W(\text{me})$ | 8 |
| 5,8 | \square | //contradiction! |

Hence $W(\text{me})$ //I win!!

[Unit 5: Machine Learning]
Artificial Intelligence (CSC 355)

Jagdish Bhatta

Central Department of Computer Science & Information Technology
Tribhuvan University

What is Learning?

"Learning denotes changes in the system that are adaptive in the sense that they enable the system to do the same task (or tasks drawn from the same population) more effectively the next time." --Herbert Simon

"Learning is constructing or modifying representations of what is being experienced." --Ryszard Michalski

"Learning is making useful changes in our minds." --Marvin Minsky

Types of Learning:

The strategies for learning can be classified according to the amount of inference the system has to perform on its training data. In increasing order we have

1. **Rote learning** – the new knowledge is implanted directly with no inference at all, e.g. simple memorisation of past events, or a knowledge engineer's direct programming of rules elicited from a human expert into an expert system.
2. **Supervised learning** – the system is supplied with a set of training examples consisting of inputs and corresponding outputs, and is required to discover the relation or mapping between them, e.g. as a series of rules, or a neural network.
3. **Unsupervised learning** – the system is supplied with a set of training examples consisting only of inputs and is required to discover for itself what appropriate outputs should be, e.g. a *Kohonen Network* or *Self Organizing Map*.

Early expert systems relied on rote learning, but for modern AI systems we are generally interested in the supervised learning of various levels of rules.

The need for Learning:

As with many other types of AI system, it is much more efficient to give the system enough knowledge to get it started, and then leave it to learn the rest for itself. We may even end up with a system that learns to be better than a human expert.

The **general learning approach** is to generate potential improvements, test them, and discard those which do not work. Naturally, there are many ways we might generate the potential improvements, and many ways we can test their usefulness. At one extreme, there are model driven (top-down) generators of potential improvements, guided by an understanding of how the problem domain works. At the other, there are data driven (bottom-up) generators, guided by patterns in some set of training data.

Machine Learning:

As regards machines, we might say, very broadly, that a machine learns whenever it changes its structure, program, or data (based on its inputs or in response to external information) in such a manner that its expected future performance improves. Some of these changes, such as the addition of a record to a data base, fall comfortably within the province of other disciplines and are not necessarily better understood for being called learning. But, for example, when the performance of a speech-recognition machine improves after hearing several samples of a person's speech, we feel quite justified in that case saying that the machine has learned.

Machine learning usually refers to the changes in systems that perform tasks associated with artificial intelligence (AI). Such tasks involve recognition, diagnosis, planning, robot control, prediction, etc. The changes might be either enhancements to already performing systems or synthesis of new systems.

Learning through Examples: (A type of Concept learning)

Concept learning also refers to a learning task in which a human or machine learner is trained to classify objects by being shown a set of example objects along with their class labels. **The learner will simplify what has been observed in an example. This simplified version of what has been learned will then be applied to future examples.** Concept learning ranges in simplicity and complexity because learning takes place over many areas. When a concept is more difficult, it will be less likely that the learner will be able to simplify, and therefore they will be less likely to learn. This learning by example consists of the idea of **version space**.

A **version space** is a hierarchical representation of knowledge that enables you to keep track of all the useful information supplied by a sequence of learning examples without remembering any of the examples.

The **version space method** is a concept learning process accomplished by managing multiple models within a version space.

Version Space Characteristics

In settings where there is a generality-ordering on hypotheses, it is possible to represent the version space by two sets of hypotheses: (1) the **most specific** consistent hypotheses and (2) the **most general** consistent hypotheses, where "consistent" indicates agreement with observed data.

The most specific hypotheses (i.e., the specific boundary **SB**) are the hypotheses that cover the observed positive training examples, and as little of the remaining feature space as possible. These are hypotheses which if reduced any further would *exclude* a *positive* training example, and hence become inconsistent. These minimal hypotheses essentially constitute a (pessimistic) claim that the true concept is defined just by the *positive* data already observed: Thus, if a novel (never-before-seen) data point is observed, it should be assumed to be negative. (i.e., if data has not previously been ruled in, then it's ruled out.)

Artificial Intelligence

The most general hypotheses (i.e., the general boundary **GB**) are those which cover the observed positive training examples, but also cover as much of the remaining feature space without including any negative training examples. These are hypotheses which if enlarged any further would *include* a *negative* training example, and hence become inconsistent.

Tentative heuristics are represented using version spaces. A version space represents all the alternative plausible **descriptions** of a heuristic. A plausible description is one that is applicable to all known positive examples and no known negative example.

A version space description consists of two complementary trees:

1. One that contains nodes connected to overly **general** models, and
2. One that contains nodes connected to overly **specific** models.

Node values/attributes are **discrete**.

Fundamental Assumptions

1. The data is correct; there are no erroneous instances.
2. A correct description is a conjunction of some of the attributes with values.

Diagrammatical Guidelines

There is a **generalization** tree and a **specialization** tree.

Each **node** is connected to a **model**.

Nodes in the generalization tree are connected to a model that matches everything in its subtree.

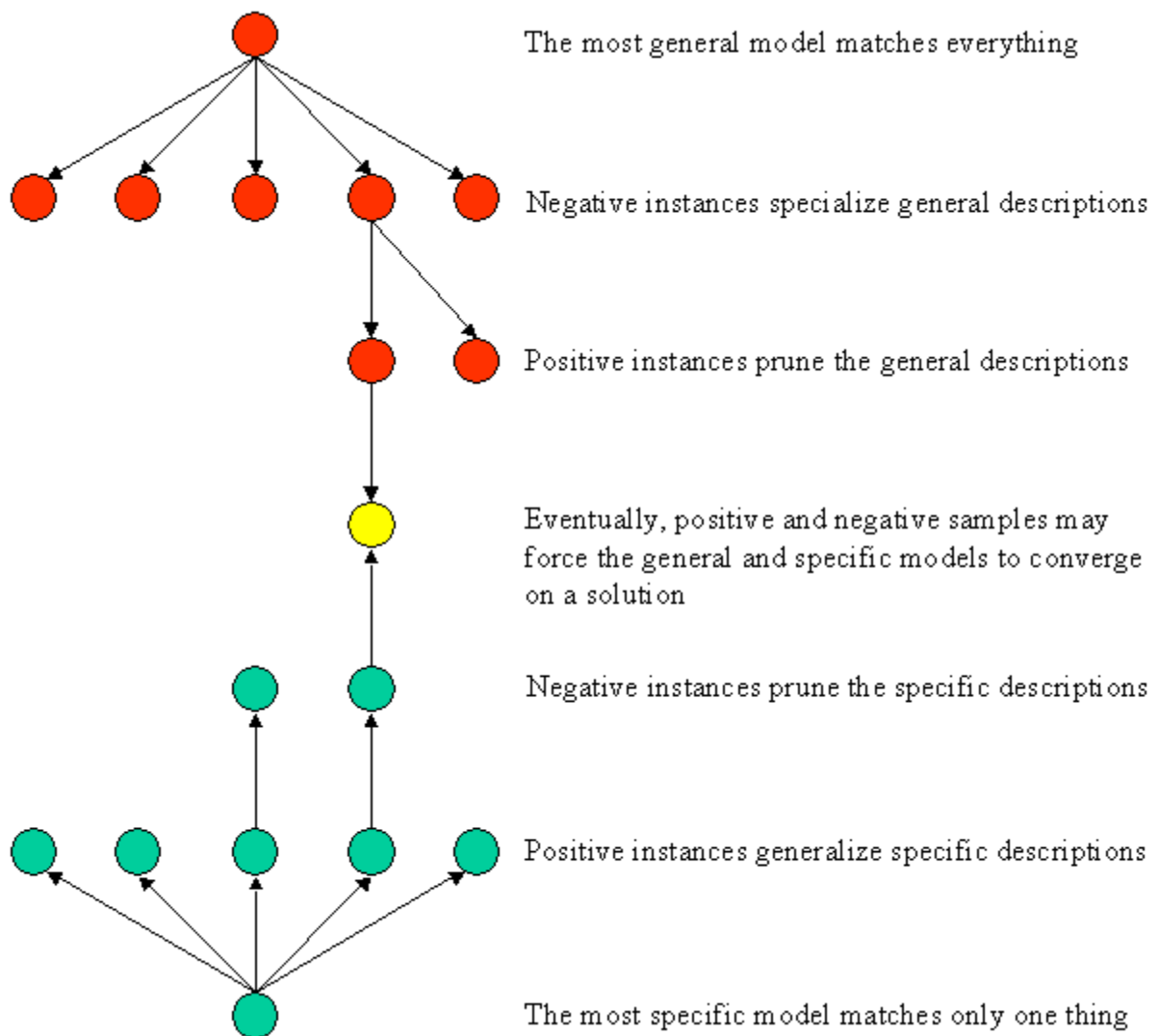
Nodes in the specialization tree are connected to a model that matches only one thing in its subtree.

Links between nodes and their models denote

- generalization relations in a generalization tree, and
- specialization relations in a specialization tree.

Diagram of a Version Space

In the diagram below, the specialization tree is colored **red**, and the generalization tree is colored **green**.



Generalization and Specialization Leads to Version Space Convergence

The key idea in version space learning is that specialization of the general models and generalization of the specific models may ultimately lead to just one correct model that matches all observed positive examples and does not match any negative examples.

That is, each time a negative example is used to specialize the general models, those specific models that match the negative example are eliminated and each time a positive example is used to generalize the specific models, those general models that fail to match the positive example are eliminated. Eventually, the positive and negative examples may be such that only one general model and one identical specific model survive.

Candidate Elimination Algorithm:

The version space method handles positive and negative examples symmetrically.

Given:

- A representation language.
- A set of positive and negative examples expressed in that language.

Compute: a concept description that is consistent with all the positive examples and none of the negative examples.

Method:

- Initialize G, the set of maximally general hypotheses, to contain one element: the null description (all features are variables).
- Initialize S, the set of maximally specific hypotheses, to contain one element: the first positive example.
- Accept a new training example.
 - If the example is **positive**:
 1. Generalize all the specific models to match the positive example, but ensure the following:
 - The new specific models involve minimal changes.
 - Each new specific model is a specialization of some general model.
 - No new specific model is a generalization of some other specific model.
 2. Prune away all the general models that fail to match the positive example.
 - If the example is **negative**:
 1. Specialize all general models to prevent match with the negative example, but ensure the following:
 - The new general models involve minimal changes.
 - Each new general model is a generalization of some specific model.
 - No new general model is a specialization of some other general model.
 2. Prune away all the specific models that match the negative example.
 - If S and G are both singleton sets, then:
 - if they are identical, output their value and halt.
 - if they are different, the training cases were inconsistent. Output this result and halt.
 - else continue accepting new training examples.

The algorithm stops when:

Artificial Intelligence

1. It runs out of data.
2. The number of hypotheses remaining is:
 - 0 - no consistent description for the data in the language.
 - 1 - answer (version space converges).
 - 2⁺ - all descriptions in the language are implicitly included.

Problem 1:

Learning the concept of "Japanese Economy Car"

Features: (Country of Origin, Manufacturer, Color, Decade, Type)

Origin	Manufacturer	Color	Decade	Type	Example Type
Japan	Honda	Blue	1980	Economy	Positive
Japan	Toyota	Green	1970	Sports	Negative
Japan	Toyota	Blue	1990	Economy	Positive
USA	Chrysler	Red	1980	Economy	Negative
Japan	Honda	White	1980	Economy	Positive

Solution:

1. **Positive Example:** (Japan, Honda, Blue, 1980, Economy)


Initialize G to a singleton set


that includes everything. $G = \{ (?, ?, ?, ?, ?) \}$

Initialize S to a singleton set $S = \{ (Japan, Honda, Blue, 1980,$

that includes the first positive Economy) }

example.

 $(?, ?, ?, ?, ?)$

$(Japan, Honda, Blue, 1980, Economy)$ 

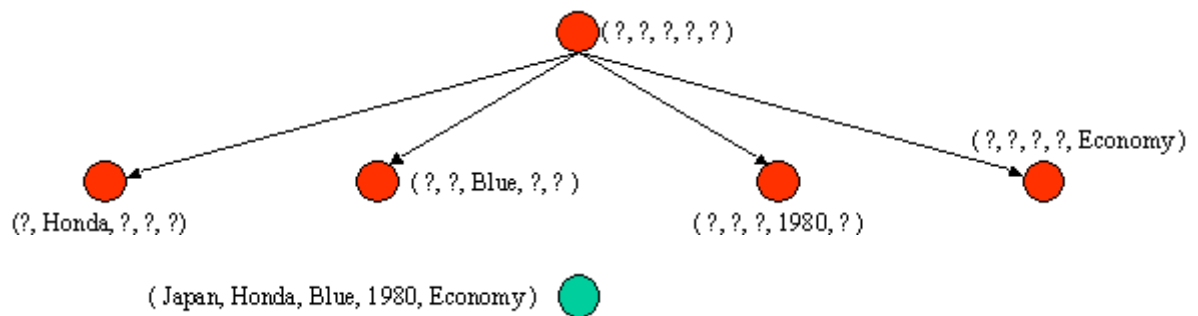
These models represent the most general and the most specific heuristics one might learn. The actual heuristic to be learned, "Japanese Economy Car", probably lies between them somewhere within the version space.

2. **Negative Example:** (Japan, Toyota, Green, 1970, Sports)

Specialize G to exclude the negative example.

$G = \{$
 $(?, \text{Honda}, ?, ?, ?),$
 $(?, ?, \text{Blue}, ?, ?),$
 $(?, ?, ?, 1980, ?),$
 $(?, ?, ?, ?, \text{Economy}) \}$

$S = \{(\text{Japan}, \text{Honda}, \text{Blue}, 1980, \text{Economy})\}$



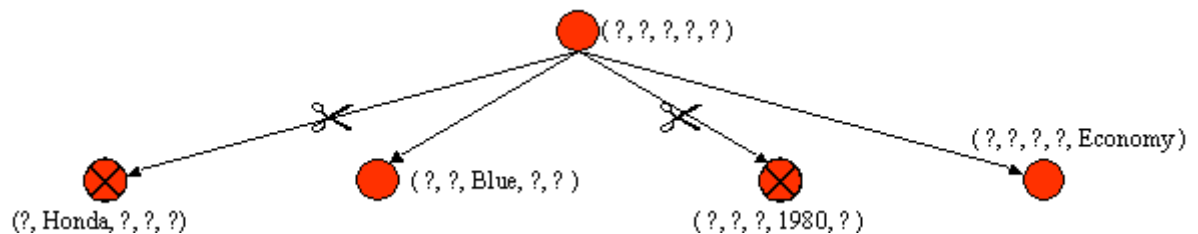
Refinement occurs by generalizing S or specializing G , until the heuristic hopefully converges to one that works well.

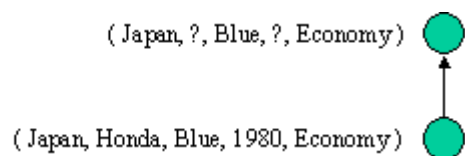
3. **Positive Example:** $(\text{Japan}, \text{Toyota}, \text{Blue}, 1990, \text{Economy})$

Prune G to exclude descriptions inconsistent with the positive example. (Prune = ✂)
 Generalize S to include the positive example.

$G = \{$
 $(?, ?, \text{Blue}, ?, ?),$
 $(?, ?, ?, ?, \text{Economy}) \}$

$S = \{(\text{Japan}, ?, \text{Blue}, ?, \text{Economy})\}$



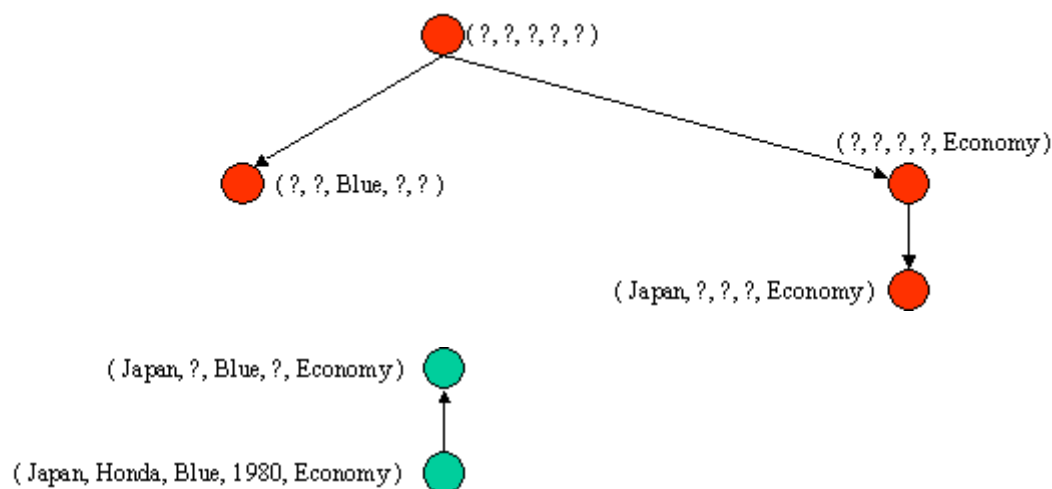


4. **Negative Example:** (USA, Chrysler, Red, 1980, Economy)

Specialize G to exclude the negative example (but stay consistent with S)

$$G = \{ (?, ?, Blue, ?, ?), \\ (Japan, ?, ?, ?, Economy) \}$$

$$S = \{ (Japan, ?, Blue, ?, Economy) \}$$



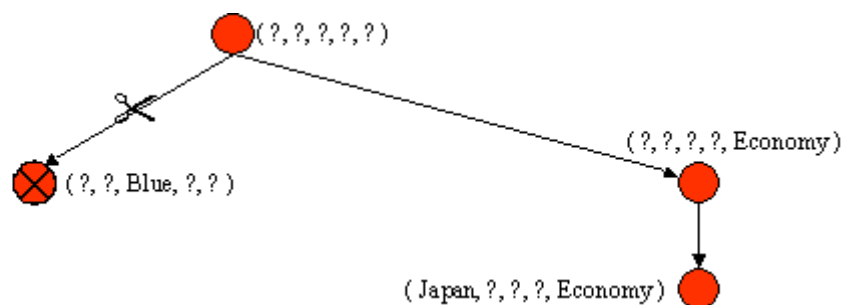
5. **Positive Example:** (Japan, Honda, White, 1980, Economy)

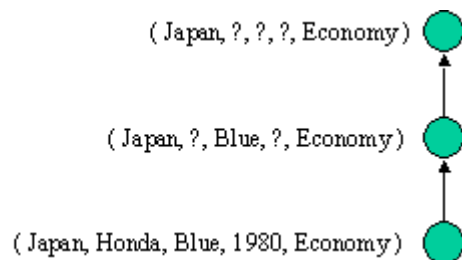
Prune G to exclude descriptions inconsistent with positive example.

Generalize S to include positive example.

$$G = \{ (Japan, ?, ?, ?, Economy) \}$$

$$S = \{ (Japan, ?, ?, ?, Economy) \}$$

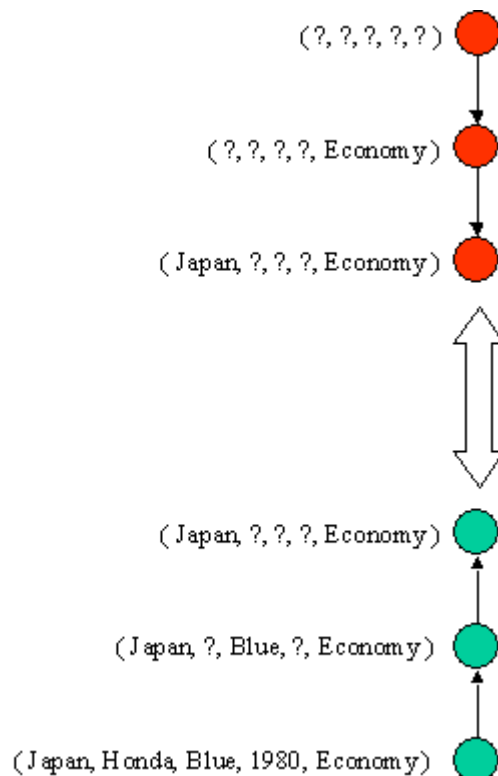




G and S are singleton sets and $S = G$.

Converged.

No more data, so algorithm stops.



Explanation Based Machine Learning:

Explanation-based learning (EBL) is a form of machine learning that exploits a very strong, or even perfect, domain theory to make generalizations or form concepts from training examples. This is a type of *analytic* learning. The advantage of explanation-based learning is that, as a deductive mechanism, it requires only a single training example (inductive learning methods often require many training examples)

An Explanation-based Learning (**EBL**) system accepts an example (i.e. a training example) and explains what it learns from the example. The **EBL** system takes only the relevant aspects of the training.

Artificial Intelligence

EBL accepts four inputs:

A training example : what the learning sees in the world. (specific facts that rule out some possible hypotheses)

A goal concept : a high level description of what the program is supposed to learn. (the set of all possible conclusions)

A operational criterion : a description of which concepts are usable. (criteria for determining which features in the domain are efficiently recognizable, e.g. which features are directly detectable using sensors)

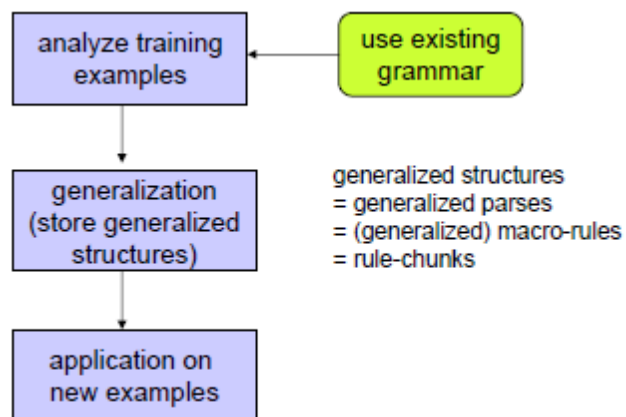
A domain theory : a set of rules that describe relationships between objects and actions in a domain. (axioms about a domain of interest)

From this EBL computes a generalization of the training example that is sufficient not only to describe the goal concept but also satisfies the operational criterion.

This has two steps:

Explanation: the domain theory is used to prune away all unimportant aspects of the training example with respect to the goal concept.

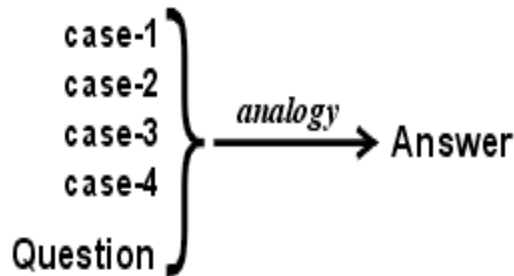
Generalisation: the explanation is generalized as far possible while still describing the goal concept



An example of EBL using a perfect domain theory is a program that learns to play chess by being shown examples. A specific chess position that contains an important feature, say, "Forced loss of black queen in two moves," includes many irrelevant features, such as the specific scattering of pawns on the board. EBL can take a single training example and determine what the relevant features are in order to form a generalization.

Learning by Analogy:

Reasoning by analogy generally involves abstracting details from a particular set of problems and resolving structural similarities between previously distinct problems. Analogical reasoning refers to this process of recognition and then applying the solution from the known problem to the new problem. **Such a technique is often identified as *case-based reasoning*.** Analogical learning generally involves developing a set of mappings between features of two instances.



The question in above figure represents some known aspects of a new case, which has unknown aspects to be determined. In deduction, the known aspects are compared (by a version of structure mapping called *unification*) with the premises of some implication. Then the unknown aspects, which answer the question, are derived from the conclusion of the implication. In analogy, the known aspects of the new case are compared with the corresponding aspects of the older cases. The case that gives the best match may be assumed as the best source of evidence for estimating the unknown aspects of the new case. The other cases show alternative possibilities for those unknown aspects; the closer the agreement among the alternatives, the stronger the evidence for the conclusion.

1. **Retrieve:** Given a target problem, retrieve cases from memory that are relevant to solving it. A case consists of a problem, its solution, and, typically, annotations about how the solution was derived. For example, suppose Fred wants to prepare blueberry pancakes. Being a novice cook, the most relevant experience he can recall is one in which he successfully made plain pancakes. The procedure he followed for making the plain pancakes, together with justifications for decisions made along the way, constitutes Fred's retrieved case.
2. **Reuse:** Map the solution from the previous case to the target problem. This may involve adapting the solution as needed to fit the new situation. In the pancake example, Fred must adapt his retrieved solution to include the addition of blueberries.
3. **Revise:** Having mapped the previous solution to the target situation, test the new solution in the real world (or a simulation) and, if necessary, revise. Suppose Fred adapted his pancake solution by adding blueberries to the batter. After mixing, he discovers that the batter has turned blue – an undesired effect. This suggests the following revision: delay the addition of blueberries until after the batter has been ladled into the pan.
4. **Retain:** After the solution has been successfully adapted to the target problem, store the resulting experience as a new case in memory. Fred, accordingly, records his newfound

procedure for making blueberry pancakes, thereby enriching his set of stored experiences, and better preparing him for future pancake-making demands.

Transformational Analogy:

Suppose you are asked to prove a theorem in plane geometry. You might look for a previous theorem that is very similar and copy its proof, making substitutions when necessary. The idea is to transform a solution to a previous problem into a solution for the current problem. The following figure shows this process,

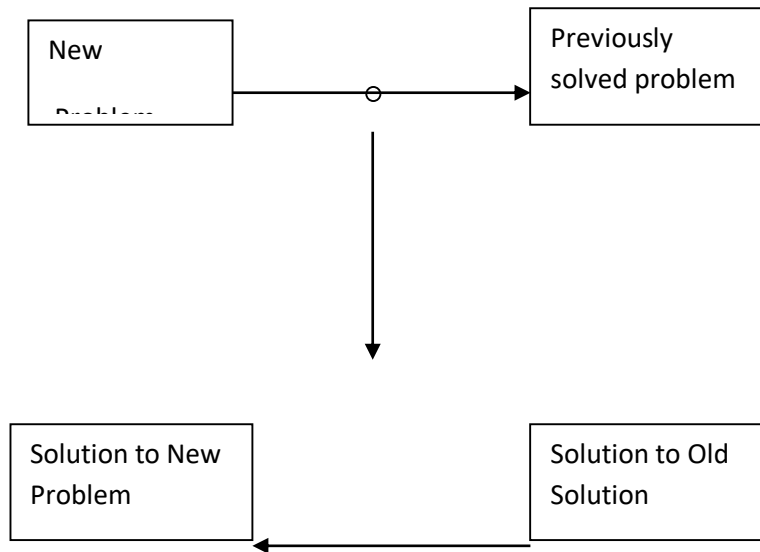


Fig: Transformational Analogy

Derivational Analogy:

Notice that transformational analogy does not look at how the old problem was solved, it only looks at the final solution. Often the twists and turns involved in solving an old problem are relevant to solving a new problem. The detailed history of problem solving episode is called derivation, Analogical reasoning that takes these histories into account is called derivational analogy.

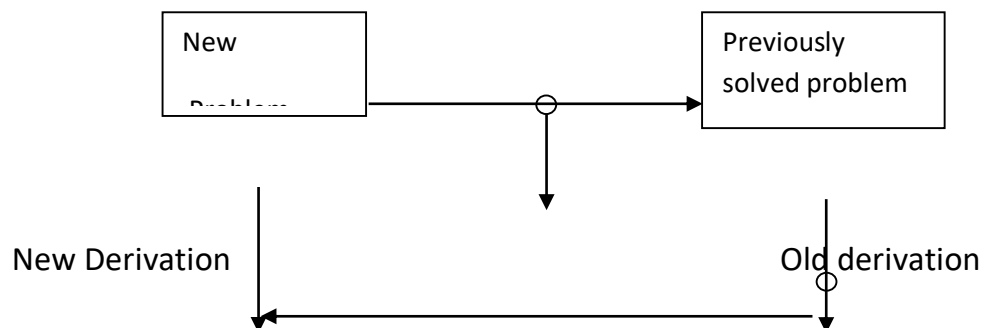




Fig: Derivational Analogy

For details of the above mentioned theory, Refer Book:- E. Rich, K. Knight, S. B. Nair, Tata MacGraw Hill (Pages 371-372)

Learning by Simulating Evolution:

Refer Book:- P. H. Winston, Artificial Intelligence, Addison Wesley. (Around page 220)

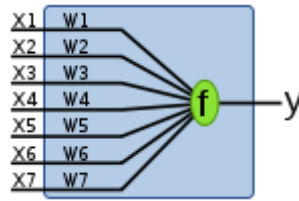
Learning by Training Perceptron:

Below is an example of a learning algorithm for a single-layer (no hidden-layer) perceptron. **For multilayer perceptrons, more complicated algorithms such as backpropagation must be used.** Or, methods such as the delta rule can be used if the function is non-linear and differentiable, although the one below will work as well.

The learning algorithm we demonstrate is the same across all the output neurons, therefore everything that follows is applied to a single neuron in isolation. We first define some variables:

- $x(j)$ denotes the j -th item in the n -dimensional input vector
- $w(j)$ denotes the j -th item in the weight vector
- $f(x)$ denotes the output from the neuron when presented with input x
- α is a constant where $0 < \alpha \leq 1$ (learning rate)

Assume for the convenience that the bias term b is zero. An extra dimension $n + 1$ can be added to the input vectors x with $x(n + 1) = 1$, in which case $w(n + 1)$ replaces the bias term.



the appropriate weights are applied to the inputs, and the resulting weighted sum passed to a function which produces the output y

Let $D_m = \{(x_1, y_1), \dots, (x_m, y_m)\}$ be training set of m training examples, where x_i is the input vector to the perceptron and y_i is the desired output value of the perceptron for that input vector.

Learning algorithm steps:

1. Initialize weights and threshold.

- Set $w_i(t)$, ($1 \leq i \leq m$) to be the weight i at time t , and ϕ to be the threshold value in the output node.
- Set $w(0)$ to be $-\phi$, the bias, and $x(0)$ to be always 1.
- Set $w_i(1)$ to small random values, thus initialising the weights and threshold.

2. Present input and desired output

- Present input $x_0 = 1$ and x_1, x_2, \dots, x_m and desired output $d(t)$

3. Calculate the actual output

- $y(t) = fh[w_0(t) + w_1(t)x_1(t) + w_2(t)x_2(t) + \dots + w_m(t)x_m(t)]$

4. Adapts weights

- $w_i(t+1) = w_i(t) + \alpha[d(t) - y(t)]x_i(t)$, for $0 \leq i \leq m$.

Steps 3 and 4 are repeated until the iteration error is less than a user-specified error threshold or a predetermined number of iterations have been completed.